UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

INGENIERÍA DE TELECOMUNICACIÓN

"SIMULADOR HÁPTICO PARA ENTRENAMIENTO DE TÉCNICAS DE LAPAROSCOPIA"

PROYECTO FIN DE CARRERA

Junio – 2008

AUTOR: Daniel Ivorra Ruiz

DIRECTOR/ES: José María Sabater Navarro

VISTO BUENO Y CALIFICACIÓN DEL PROYECTO

Título proyecto:

Simulador háptico para entrenamiento de técnicas de laparoscopia

Proyectante: Daniel Ivorra Ruiz

Director/es: José María Sabater Navarro

VºBº director/es del proyecto: Fdo.: José María Sabater Navarro Fdo.:

Lugar y fecha:

CALIFICACIÓN NUMÉRICA ☐ MATRÍCULA DE HONOR ☐

Méritos justificativos en el caso de conceder Matrícula de Honor:

Conforme presidente: Fdo.:	Conforme secretario: Fdo.:	Conforme vocal: Fdo.:

Lugar y fecha: _____

Índice

Índice 6
Resumen 7
Capítulo 1. Introducción a la simulación de técnicas quirúrgicas. 8
La cirugía laparoscópica. 9
Problema general de la simulación. 12
Estado del arte de la simulación quirúrgica 13
Requisitos del simulador de laparoscopia 14
Capítulo 2. Modelado de herramientas para laparoscopia. 15
Revisión de las herramientas actuales 15
Modelado de herramientas tipo A (punteros, con muñeca articulada y fijas) 23
Modelado de herramientas tipo B (pinzas, con muñeca articulada y fijas) 25
Otras herramientas 27
Capítulo 3. Arquitectura del simulador quirúrgico de VR2. 28
Introducción a las librerias VTK 28
Algoritmo de reconstrucción 3D utilizado (marching cubes) 29
Algoritmo de colisiones utilizado 37
Capítulo 4. Detección de colisiones. 42
Introducción a las librerías V-Collide 43
Integración de VTK y Vcollide 44
La clase CollisionObjectFactory 49
Capítulo 5. Creación de herramientas 3D con VTK 54
Introducción 54
Diseño de herramientas con 3DS Max. 54
Almacenado de las piezas en un formato adecuado. 58
Desarrollo de código para simular la herramienta. 59
Herramientas Articuladas 59
Herramientas Fijas 65
La clase Herramienta3D 72
Capítulo 6. Simulador LAPUMH 80
Interfaz Visual: QT con VTK 80
Aspecto Interfaz 80
Herramientas software 82
Capítulo 7. Conclusiones y trabajo futuro. 87
Conclusiones 87
Trabajo Futuro 88
ANEXOS 89
Anexo A: Configuración del entorno para depurar 89
Anexo B: Integración de VTK y VCollide 98
Anexo C: Manual de usuario de V-Collide 106
Anexo D: Bibliografía 111

Resumen

El proyecto realizado se engloba dentro del proyecto de robótica quirúrgica desarrollado por el grupo de Robótica y Realidad Virtual perteneciente al Departamento de Ingeniería de Sistemas Industriales de la Universidad Miguel Hernández de Elche.

El objetivo de dicho grupo es el desarrollo de un sistema de telecirugía en el que a partir de imágenes biomédicas captadas por un aparato de diagnóstico tipo CT (Tomografía Computerizada) se generen cuerpos 3D de los órganos deseados, permitiendo interactuar con los mismos mediante un interfaz háptico. Mediante el sistema descrito anteriormente, el usuario será capaz de planificar operaciones e incluso de realizarlas en tiempo real mediante un sistema robótico desarrollado también por este grupo.

El principal objetivo de este proyecto es el estudio de las librerías VTK orientándolo a la creación de herramientas 3D para la intervención laparoscópica y su integración con un sistema de detección de colisiones que posibilite la implementación del simulador de laparoscopia.

Capítulo 1.
Introducción a la simulación de técnicas quirúrgicas.

Los avances que ha sufrido el campo de la tecnología de la información y, dentro del mismo, la Informática Gráfica, ha permitido el comienzo de la investigación y desarrollo de simuladores virtuales para el entrenamiento de especialistas. Uno de las principales aplicaciones de estos avances es la creación de Simuladores Quirúrgicos. Este tipo de simuladores se caracteriza por sus grandes exigencias tanto visual como del sentido del tacto, sentidos fundamentales en cirugía. Para conseguir el realismo visual, los órganos presentes en el simulador se han de mostrar lo más similarmente posible a como se observan en la realidad y han de responder en tiempo-real (al menos 15 veces por segundo) a las interacciones del usuario. Respecto al realismo en el sentido del tacto, dada la elevada resolución que el ser humano posee en este sentido, el usuario del simulador precisa de una frecuencia de refresco de este sentido de al menos 500 veces por segundo. Actualmente, los grupos de investigación están dedicando grandes esfuerzos en conseguir alcanzar estos objetivos de realismo con el propósito de obtener una herramienta extremadamente potente que revolucione la docencia, la práctica y la difusión de las técnicas quirúrgicas.

La simulación de técnicas quirúrgicas en el marco de la enseñanza de medicina está adquiriendo cada vez más importancia respecto a los métodos de enseñanza tradicionales. Los simuladores ofrecen una experiencia de aprendizaje estructurada, permitiendo practicar sin dañar a los pacientes. Además, facilitan la enseñanza de casos inusuales y pueden utilizarse para conseguir un nivel determinado de habilidades.

La simulación es una parte integrada en la educación quirúrgica. Los cirujanos desarrollan y adquieren habilidades a través de la práctica. Para aprender las técnicas se utilizan animales, cadáveres y pacientes. Esto tiene muchas desventajas ya que los animales tienen una anatomía diferente, los cadáveres no tienen la respuesta fisiológica adecuada y en el caso de pacientes, existe un riesgo de dañarlos al estar practicando con ellos un aprendiz.

De igual forma, el campo de la cirugía, ha experimentado avances dando lugar a un tipo de cirugía: la cirugía minimamente invasiva.

La cirugía minimamente invasiva es una técnica que permite intervenciones a través de incisiones muy pequeñas. Este procedimiento minimiza el trauma del paciente y requiere períodos de rehabilitación más cortos con respecto a la cirugía abierta. Sin embargo, debido a que la visibilidad y la operabilidad de las herramientas es reducida, constituye un procedimiento quirúrgico complejo.

Si el entrenamiento es esencial en cualquier tarea especializada, en el entrenamiento quirúrgico es más crítico ya que un error puede producir severos daños en el paciente pudiendo llegar a provocar incluso su muerte.

Además, el modelo actual de enseñanza quirúrgica es una enseñanza por oportunidad, es decir, si un paciente tiene apendicitis, los estudiantes pueden aprender aspectos técnicos de la apendicetomía. Con el uso de simuladores quirúrgicos se pueden generar diferentes escenarios, y además incrementando la complejidad de forma gradual. De esta forma, las intervenciones se pueden practicar de forma segura en un simulador antes de operar a un paciente.

La cirugía laparoscópica.

Las cirugías abdominales tradicionalmente han supuesto largas incisiones, de tal forma que permitieran al cirujano tener un campo visual para realizar la operación. Sin embargo, este método ha contribuido significativamente a la lenta recuperación de los pacientes.

Los avances en cirugía han creado nuevas técnicas y el material quirúrgico adecuado para evitar los inconvenientes debidos a grandes incisiones, dando lugar a la cirugía minimamente invasiva.

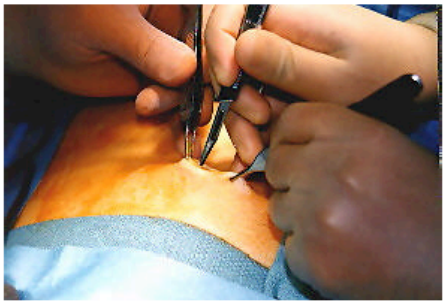
La cirugía minimamente invasiva reduce el tamaño de la incisión y la estancia en hospital de tal forma que los pacientes pueden volver a su actividad normal en un periodo de tiempo más corto. Los nuevos dispositivos permiten a los cirujanos realizar muchas operaciones sin el trauma y el dolor de las primeras técnicas.

La cirugía laparoscópica es una técnica mediante la cual se realizan únicamente pequeñas incisiones de aproximadamente 1 cm de diámetro. A diferencia de la cirugía tradicional, en lugar de mirar directamente al órgano del cuerpo que está siendo tratado, los médicos monitorizan el procedimiento a través de una videocámara especial llamada laparoscopio. Esta cámara se introduce a través de una de los pequeños agujeros, mientras los otros instrumentos son manipulados desde los otros puntos de incisión.

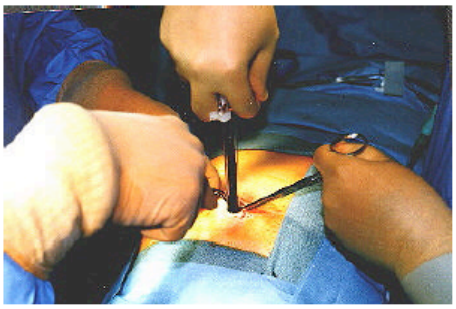
Contrariamente a la cirugía "abierta" donde los cirujanos pueden manipular y palpar (tocar) el tejido u órgano durante la operación, cualquier procedimiento efectuado en la laparoscopia es llevado a cabo utilizando un instrumento laparoscópico diseñado para encajar dentro del cuerpo a través de unas guías llamadas trocar, que entra a través de la pequeña incisión. El objetivo de este diseño es transformar el movimiento de la mano del cirujano a través de un largo tubo de pequeño diámetro creando una o más funciones de salida en la punta opuesta, dentro de la cavidad del cuerpo.

a) Intervención de laparoscopia: los cirujanos y sus ayudantes observan el campo operatorio a través de un monitor de vídeo 2D.

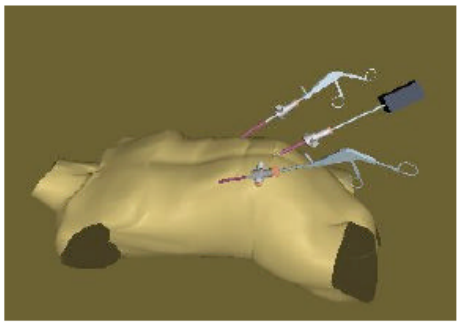
b) Instrumental quirúrgico para una intervención de cirugía laparoscópica.



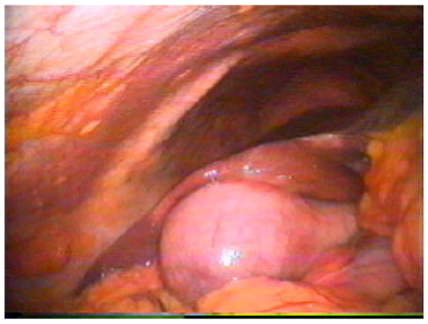
c) Creación de un pequeño corte a través del que se introducirá un instrumento quirúrgico.



d) Introducción del instrumental quirúrgico.



e) Típicos puntos de entrada de las herramientas en cirugía laparoscópica.



f) Visión del campo operatorio mediante la cámara introducida por uno de los cortes practicados en la piel.

Problema general de la simulación.

Un buen simulador quirúrgico debe de cumplir una serie de características: debe mostrar los órganos internos de manera realista, que estos órganos respondan a las interacciones con el cirujano, por ejemplo deformándose y que respondan mediante modificaciones estructurales a acciones quirúrgicas habituales como cortes, cauterización o sutura.

El problema actual es que conforme los simuladores son más complejos la imagen y la interrelación con el usuario es menos realista y no traducen el comportamiento real de los tejidos a nuestra actuación ya que estos simuladores requieren ordenadores y programas informáticos más grandes y complejos.

En resumen, los principales problemas de la simulación:

- Elevado coste computacional para simular la detección de colisiones, la deformabilidad, etc..
- Elevada dificultad para lograr realismo

Estado del arte de la simulación quirúrgica

Los simuladores quirúrgicos imponen dos restricciones a los entornos: realismo virtual y físico. La primera requiere que los órganos se muestren lo más fielmente posible a cómo aparecen en la cirugía real. La segunda requiere que el comportamiento de los órganos sea lo más similar posible al de los órganos reales. La complejidad de este comportamiento excede la capacidad de cálculo de los ordenadores convencionales, requiriendo alcanzar una solución de compromiso entre realismo y prestaciones.

Actualmente con los simuladores virtuales se pueden realizar prácticamente la mayoría de los ejercicios básicos (coordinación mano-vista; corte, cauterización, anudado, etc.) sin embargo quedan por desarrollar ejercicios complejos que requieren ordenadores de mayor potencia. No obstante, se están dando los pasos necesarios para que en los próximos años podamos disponer de simuladores que nos permitan reproducir alguno de los procedimientos quirúrgicos que ahora requieren el empleo de animales de experimentación.

Requisitos del simulador de laparoscopia

Para un realismo visual, se precisa una frecuencia de refresco de entre 20 Hz y 60 Hz. Por otro lado, dada la mayor resolución en sensación de tacto que poseen los seres humanos, éstos precisan de una frecuencia de refresco de entre 300 Hz y 1000 Hz para tener una sensación realista en este tipo de realimentaciones.

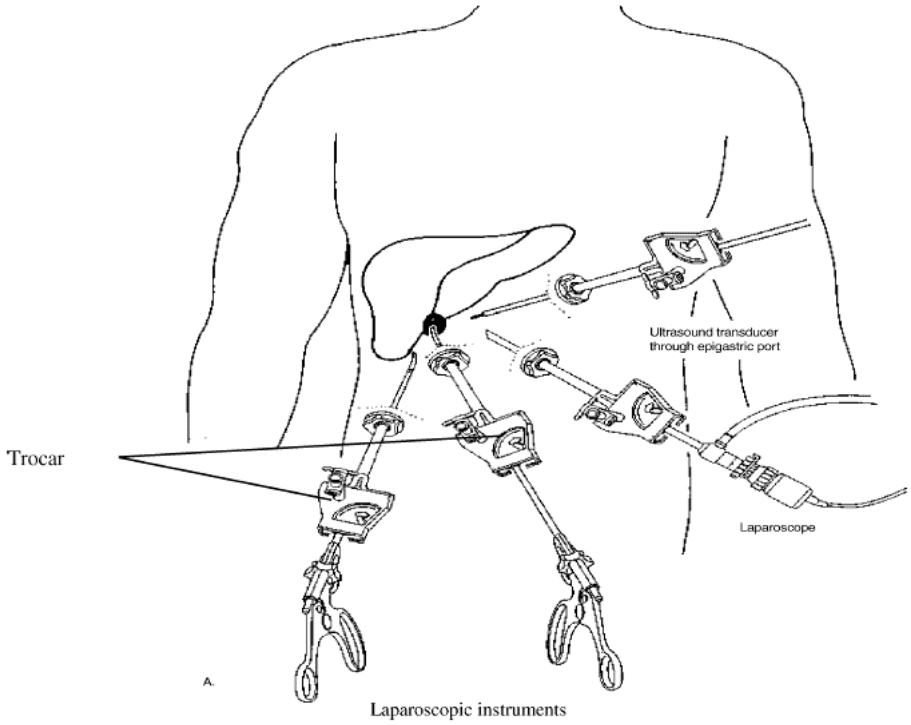
Por otro lado, para este proyecto, se trabajará con órganos rígidos, es decir, no deformables, y se requerirá la detección de colisiones entre los órganos y las herramientas de laparoscopia.

Capítulo 2. Modelado de herramientas para laparoscopia.

Revisión de las herramientas actuales

Un mecanismo es un dispositivo mecánico que se compone de piezas conectadas unas con otras mediante uniones de tal forma que las piezas pueden moverse mediante un movimiento relativo controlado. Muchas herramientas de laparoscopia son muy complejas y tienen multiples componentes mecánicos interactuando. Estas herramientas cada vez son más complejas con el objetivo de conseguir más funciones de salida a partir de una sencilla acción de entrada.

Es debido a esta complejidad en los mecanismos de ensamblado que se incrementa el coste de fabricación a causa de tener más partes y del tiempo empleado en conseguir un diseño apropiado.



Para caracterizar una herramienta de laparoscopia podemos fijarnos en los diferentes atributos del sistema mecánico: entrada, mecanismos intermedios, salidas, diseño y activación.

La entrada representa el método que empleamos para activar el instrumento y puede ser de fuerza autocontenida o manual. Autocontenida describe un instrumento que tiene su propia fuente de energía para accionar el dispositivo, mientras manual hace referencia a las manos del cirujano como origen de la fuerza.

Las entradas y salidas no están directamente conectadas. Por ejemplo, en la grapadora quirúrgica de la siguiente imagen, los mecanismos dentro de la empuñadura son componentes del mecanismo intermedio.

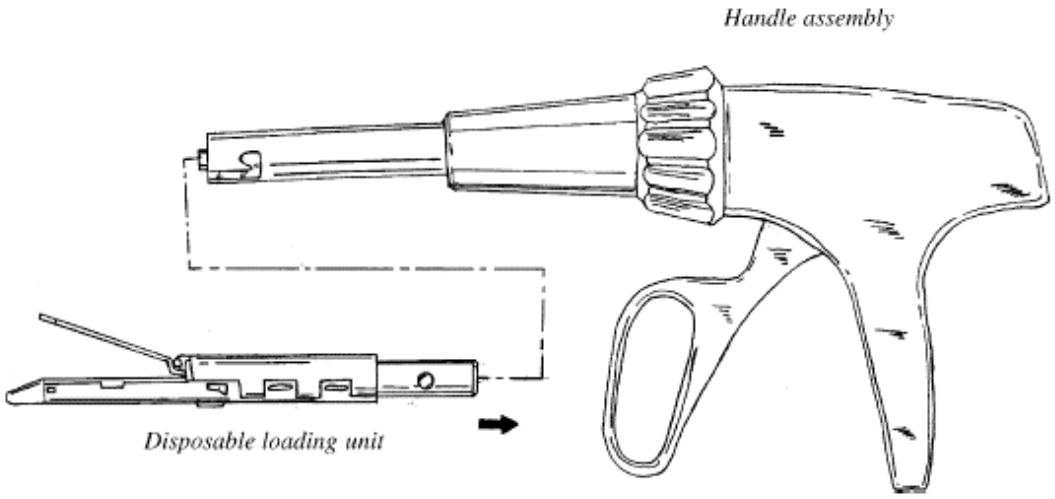


Fig. 2. Laparoscopic surgical stapler [8].

Los mecanismos intermedios representan todos los componentes mecánicos necesarios para que funcione la grapadora, es decir, los mecanismos que se encargan de recibir la acción de entrada y transformarla en una acción de salida útil.

En el ejemplo de la grapadora, la entrada se realiza a través de la palanca de accionamiento y es transmitida al armazón del engranaje a través del linguete de trinquete y después al mango de accionamiento antes de que las funciones de salida de grapar, expulsar las grapas y suturar puedan ser realizadas por la unidad de carga desechable (Disposable loading unit). Esta herramienta se utiliza en operaciones de laparoscopia en las que se necesita eliminar una sección enferma. La herramienta se utiliza en un extremo del órgano aplicando filas paralelas de grapas al tejido para que deje de sangrar, mientras una cuchilla irá cortando el tejido u órgano entre las filas de grapas. A continuación, el instrumento se recarga con un nuevo cartucho de grapas. Se aplica entonces al otro extremo del tejido u órgano para que la sección enferma esté totalmente diseccionada y los extremos restantes del órgano permanecen sellados. Para retirar la sección enferma del órgano se utiliza otra herramienta.

En cuanto a las acciones de salida de las herramientas de laparoscopia, un ejemplo de las más comunes son: rotar, virar, sujetar, expulsar o eyectar, grapar y cortar. Muchas de ellas son diseñadas en una única herramienta, por ejemplo, en la grapadora, por razones ergonómicas para que los cirujanos puedan, sin esfuerzo, utilizar la herramienta en una

posición cómoda. Por ejemplo, para dirigir la grapadora hacia la localización donde tiene que grapar el tejido, el cirujano puede utilizar la capacidad de viraje y rotación. Como parte de la operación, el instrumento debe sujetar el tejido para después grapar los tejidos (las partes superior e inferior del tejido) y cortar entre las grapas (esto sólo en algunas herramientas). Para realizar estas funciones las herramientas disponen de ciertos mecanismos en los que no vamos a entrar en detalle. A modo de ejemplo, las tareas de sujetar, expulsar las grapas y el cortar el tejido se realizan simultáneamente por un tornillo mecanizado y un deslizador mecánico conectados con un motor eléctrico.

La siguiente figura representa una herramienta dotada de motor eléctrico con su propia fuente de energía (pilas)) para activar el ensamblado del motor. Cuando se acciona el gatillo el engranaje motorizado se activa, actuando a su vez sobre un engranaje axial sujetado sobre un mecanismo que aplicará las grapas sobre el tejido.

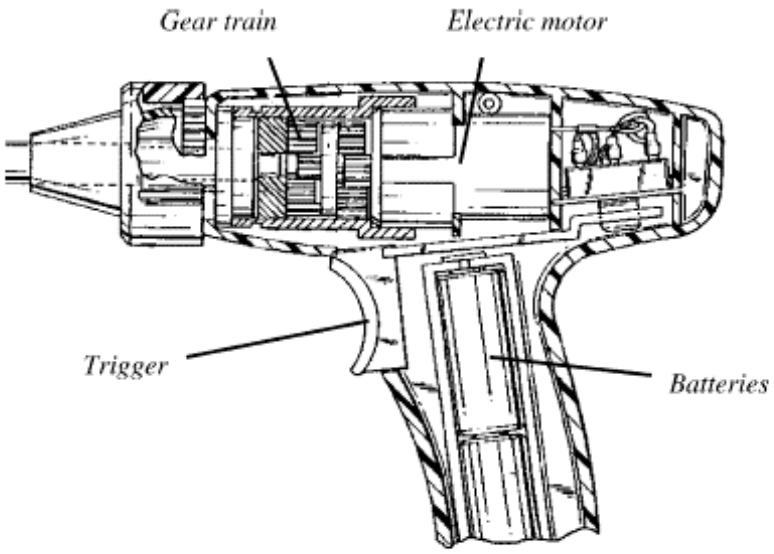


Fig. 4. Electric motor powered [14].

En cuanto a la activación, se entiende por activación al control del movimiento de entrada requerido para conseguir las funciones de salida deseadas. En herramientas de energía autocontenida los cirujanos tienen poco o nada de control sobre la salida una vez que han apretado el botón o disparador. En herramientas de entrada manual, los cirujanos normalmente pueden controlar la velocidad con la que se va a realizar la función de salida, especialmente para eyectar las grapas y cortar el tejido. Controlando la velocidad con la que

se apreta el gatillo (o cualquier otro tipo de disparador), el cirujano puede determinar la velocidad con la que se eyectarán las grapas y/o se cortará el tejido.

La siguiente figura representa una grapadora quirúrgica que funciona manualmente. El mango de actuación está montado mediante un pivote, de tal forma que mediante un mecanismo de rosca y muelle, controla un movimiento lineal del eje de actuación para desplegar las grapas.

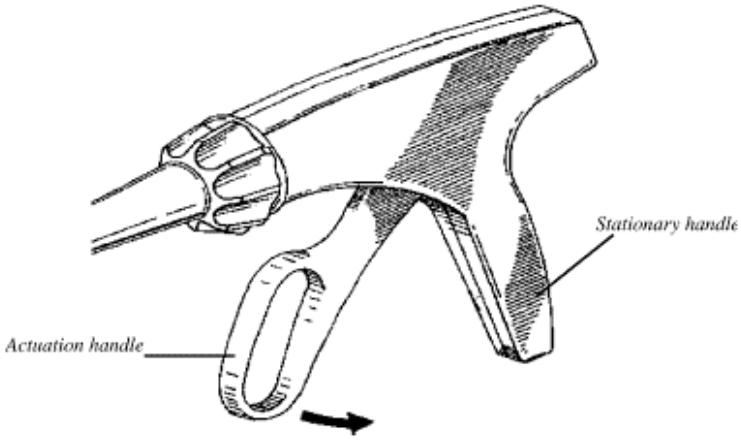


Fig. 6. Rotational input [8].

La figura siguiente también es una grapadora activada manualmente. Esta herramienta se acciona mediante un mecanismo de embudo o entrada deslizante (tipo bombín). Se trata de un tubo, que al hacer fuerza longitudinalmente hace que al otro extremo se desplieguen las grapas.

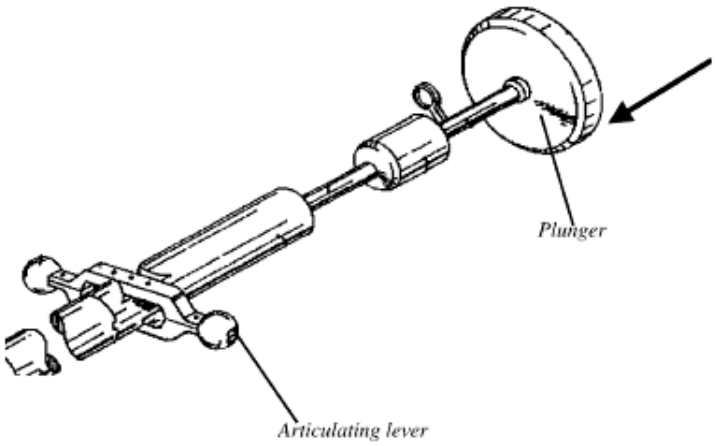


Fig. 7. Sliding input [3].

En la siguiente figura se observa las formas comunes de maniobrar una grapadora quirúrgica. Además del grado de libertad que actúa la mordaza este dispositivo tiene dos grados adicionales de libertad en articulación: rotación y viraje (movimiento a izquierda y derecha). Estos movimientos se controlan mediante dos medios diferentes: la palanca de control (control lever) y el collar de rotación (rotation collar). Este último se utiliza para un movimiento rotacional y el primero para un movimiento a izquierda y derecha.

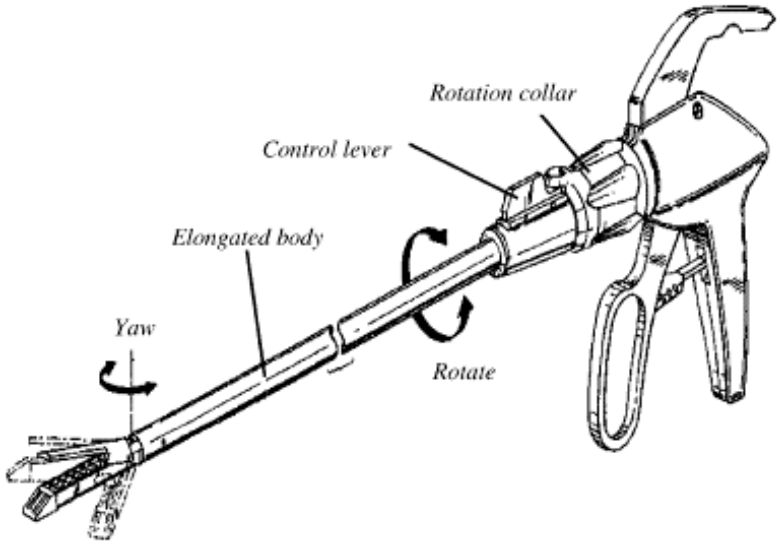


Fig. 8. Instrument's maneuverability [5].

La siguiente figura muestra una vista de un corte de sección lateral de la mandíbula superior de una grapadora quirúrgica, que está sujetando el tejido mientras las grapas están

desplegándose desde el cartucho. En ella se pueden ver las grapas en su forma parcialmente formada y final.

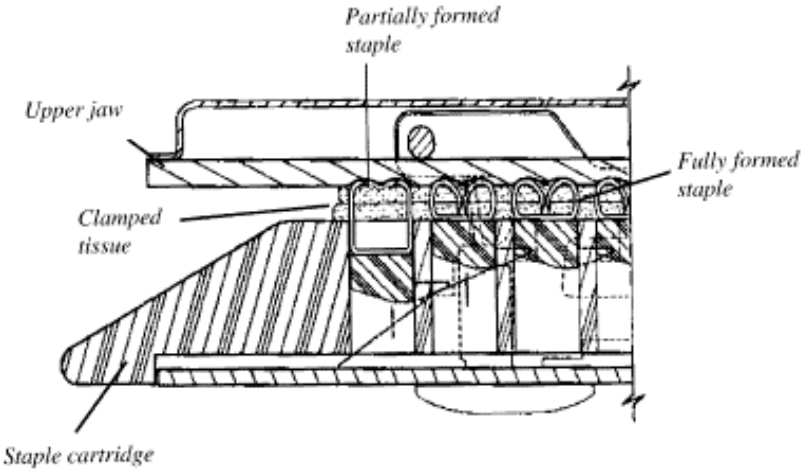


Fig. 9. Distal end during clamping and stapling [9].

Algunas herramientas de grapado quirúrgicas tienen la habilidad de cortar el tejido después de aplicar las grapas al tejido. En la siguiente figura se puede apreciar un cartucho de grapas con un hueco longitudinal para recibir una cuchilla.

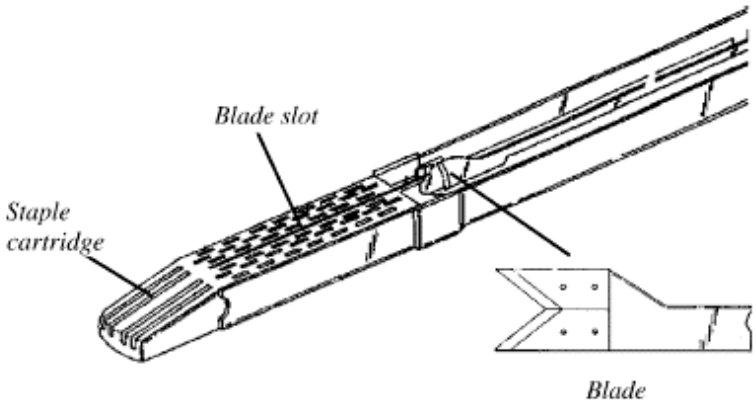


Fig. 10. Tissue cutting [15].

Para cumplir con su funcionalidad, las herramientas de laparoscopia utilizan componentes mecánicos convencionales como cables, poleas, muelles, etc… A modo de ejemplo las siguientes figuras muestran los componentes internos de dos grapadoras.

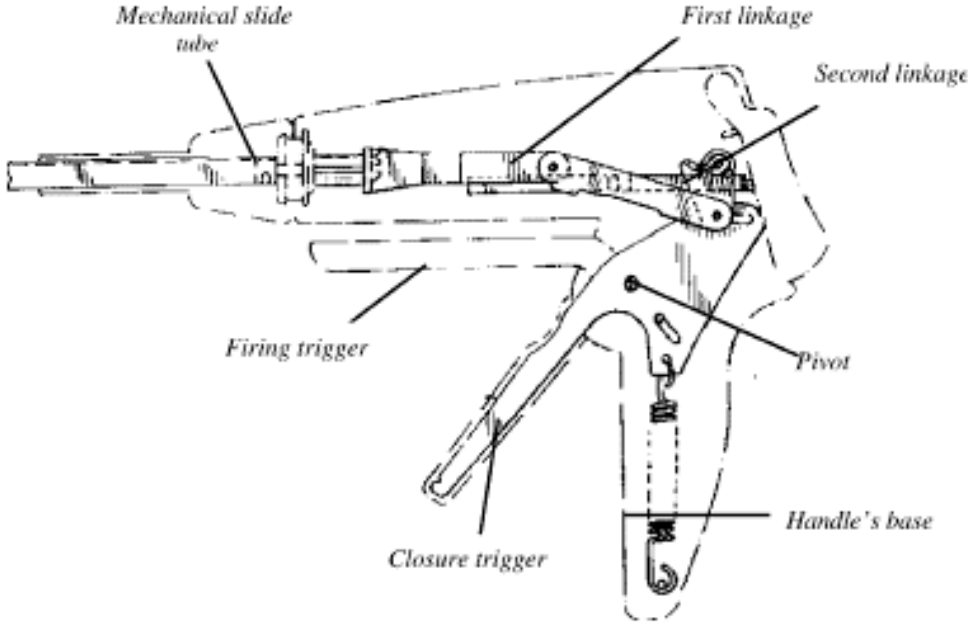


Fig. 15. A linkage solution [13].

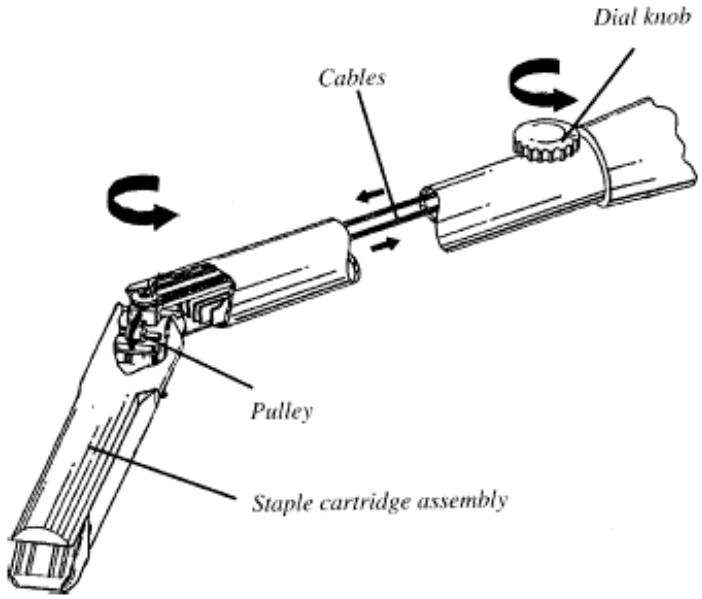
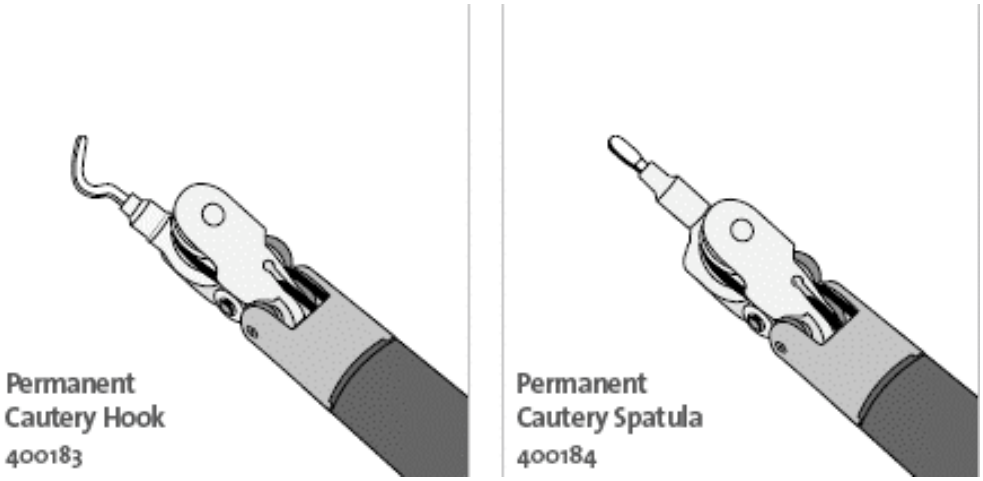


Fig. 17. Cables and pulley [4].

Modelado de herramientas tipo A (punteros, con muñeca articulada y fijas)

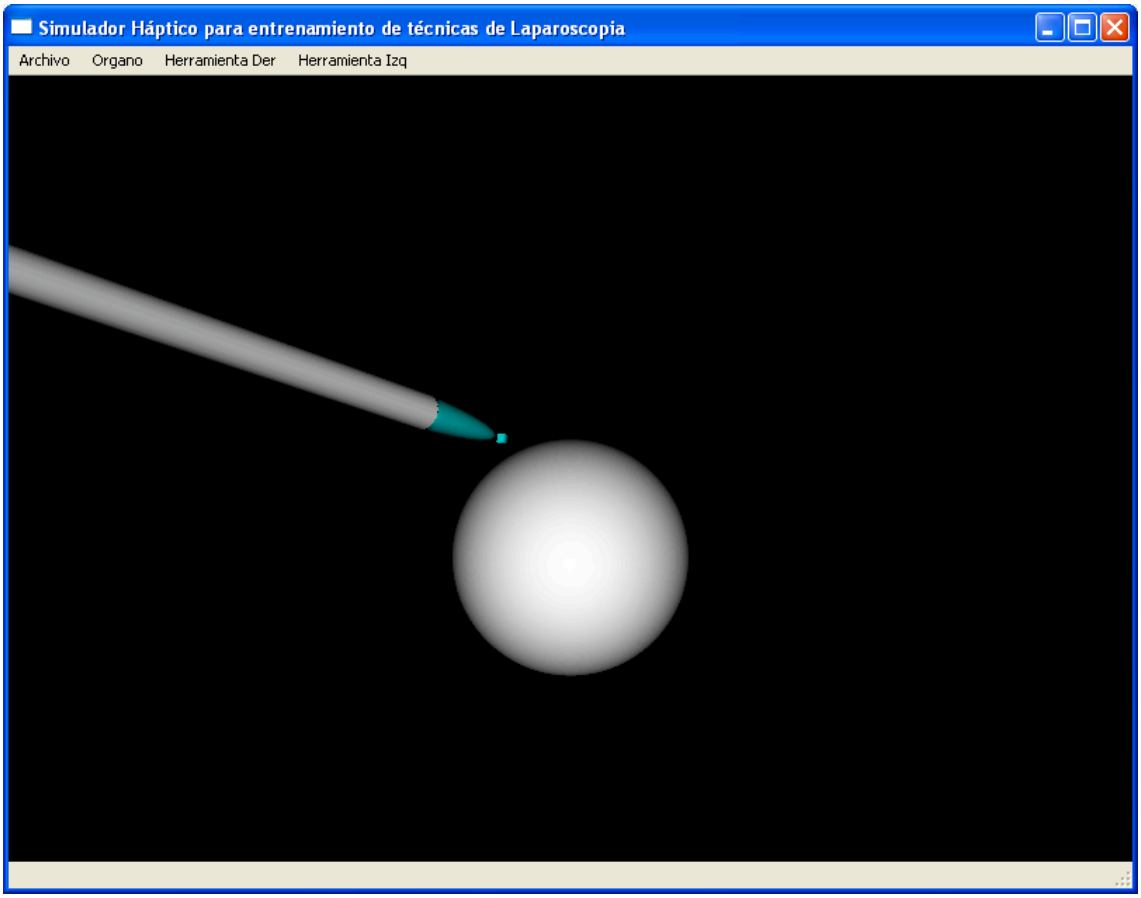
Las principales herramientas de tipo A son las herramientas de cauterización1.

En la figura siguiente se muestran dos herramientas de este tipo, una en forma de gancho y la otra en forma de espátula.



En este proyecto se ha diseñado una herramienta lápiz que se podría utilizar tanto como de lápiz como de herramienta de cauterización. Se trata de una modelo sencillo compuesto de un tubo y de una semiesfera alargada, tal y como se puede apreciar en la siguiente figura. El objeto que aparece en la punta de la herramienta es un cubo y se puede utilizar como zona de interacción de la herramienta. Por ejemplo, para cauterizar sólo a través de la punta.

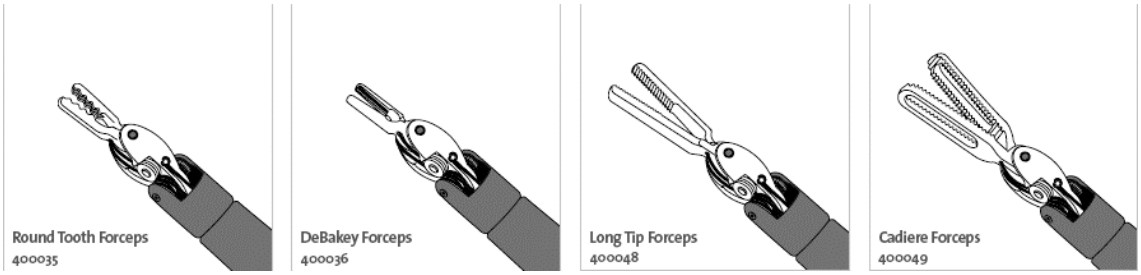
[1] Es un término medico usado para describir la quemadura del cuerpo usada para remover una parte de él o para sellar vasos sanguíneos.


Simulador Háptico para entrenamiento de técnicas de Laparoscopia
Archivo
Organo
Herramienta Der
Herramienta Izq

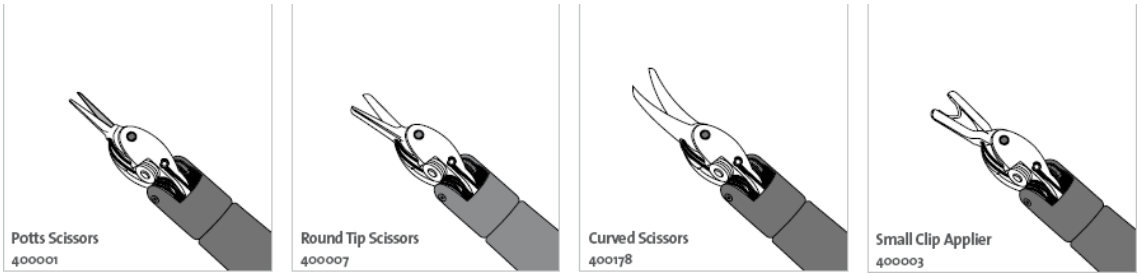
Modelado de herramientas tipo B (pinzas, con muñeca articulada y fijas)

Se entiende por herramientas de tipo B aquellas que en su extremo tienen dos componentes, como por ejemplo tijeras, pinzas y grapadoras.

En la figura se muestran varias herramientas de tipo pinza:



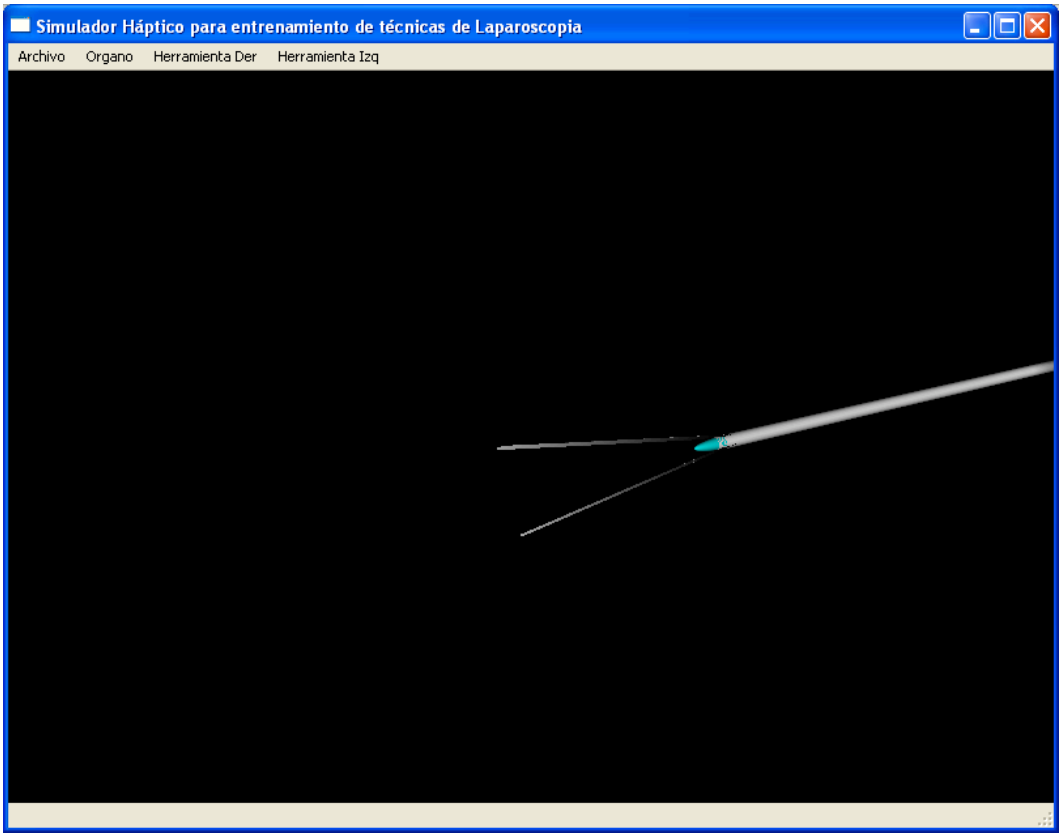
En la siguiente figura se muestran algunos ejemplos de tijeras:



En este proyecto se ha diseñado una herramienta grapadora que se podría utilizar tanto como de pinzas como de herramienta grapadora e incluso como herramienta de cauterización. Se trata de una modelo sencillo compuesto de un tubo y de una semiesfera alargada, tal y como estaba hecho el lápiz, pero además se compone de dos ortoedros para formar la parte de grapadora (o de pinza).

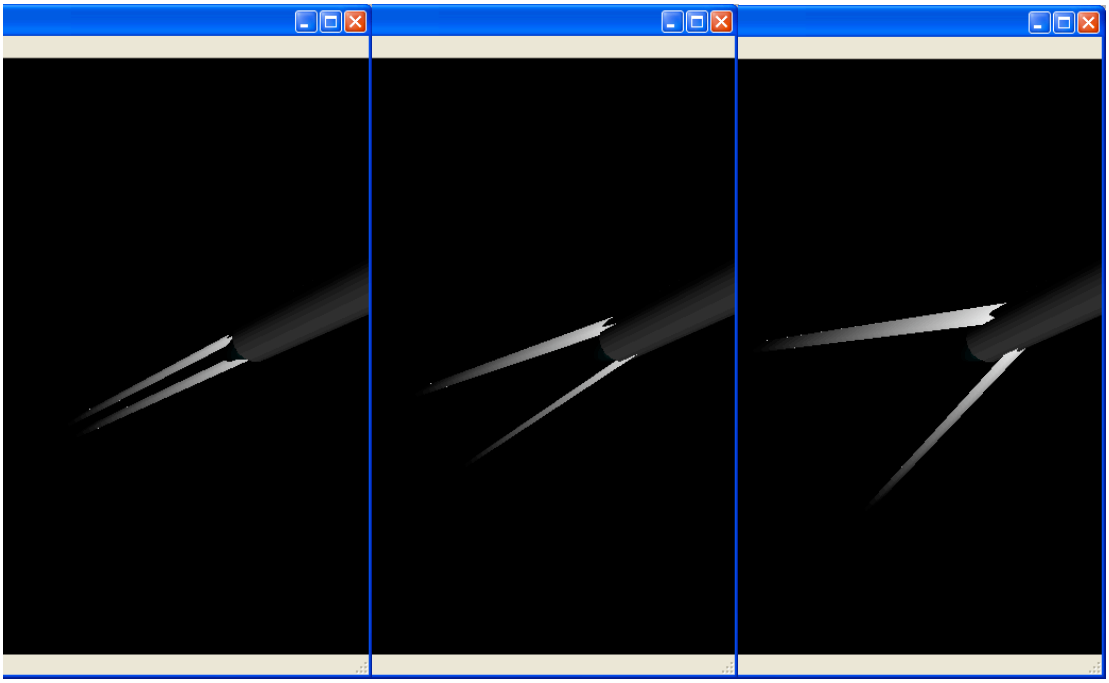
Para crear la articulación se ha hecho uso de objetos de tipo Ensamblado[2].

[2] Para más información consultar la clase vtkAssembly de las librerías vtk.



En estas herramientas se pueden realizar acciones como por ejemplo:

- abrir/cerrar las pinzas
- grapar
- rotar las pinzas



Otras herramientas

Además de las herramientas vistas en los apartados anteriores existen otras herramientas como herramientas para coser, herramientas de tubo flexible, etc.. pero que no han sido objeto de estudio en este proyecto.

De todas formas, si se quisieran realizar, teniendo en cuenta los avances de este proyecto, su complejidad se resumiría en el diseño de las partes de la herramienta mediante una herramienta gráfica de diseño 3D.

Capítulo 3. Arquitectura del simulador quirúrgico de VR2.

Introducción a las librerias VTK

VTK (Visualzation Toolkit) es un conjunto de librerías desarrollado por Kitware consistente en un sistema de visualización por software que permite la visualización de geometrías 3D, soportando una amplia variedad de algoritmos de visualización y modelado. El ser una herramienta open-source ha impulsado una gran difusión de la misma, extendiendo su aplicación a prácticamente todos los campos en los que se emplean objetos 3D entre los que destacan la medicina, las herramientas industriales…

Las funciones que integran VTK se basan en los principios de orientación a objetos conformando un completo sistema jerárquico de filtros y fuentes que dotan de gran flexibilidad a todo el sistema.

El modelo gráfico de VTK posee un nivel de abstracción mayor que el de otras librerías de renderizado como OpenGL o PEX, lo que se traduce en una mayor sencillez a la hora de la implementación de aplicaciones gráficas por parte del programador.

VTK no consiste únicamente en un sistema de visualización sino que aporta una amplia gama de algoritmos para tratar las imágenes y objetos creados, permitiendo entre otras cosas el añadir texturas, aplicar tensores, reducir el número de polígonos... además de proporcionar una gran variedad de clases de datos para trabajar (datos poligonales, imágenes, volúmenes, mallas…)

Algoritmo de reconstrucción 3D utilizado (marching cubes)

Creación de modelos 3D.

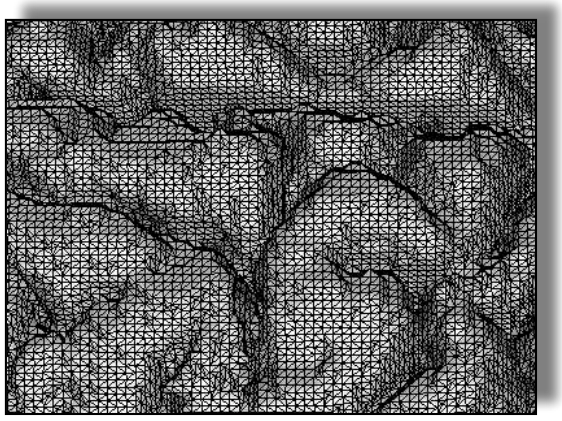
Para la creacion de modelos 3D a partir de datos volumétricos se ha optado en la mayoría de los casos por un sistema manual utilizando procesamiento de imágenes. Este proceso puede ser muy lento además de requerir conocimientos de tipo medico y mucha experiencia. Por razones obvias, es preferible contar con un proceso de segmentacion automática que permita la reconstruccion del modelo 3D de forma rapida y con la mayor precision. En el caso de determinadas partes del cuerpo, como pueden ser los huesos o la piel,el cambio de intensidad en las imagens es tan claro, que permite utilizar tecnicas de reconstruccion de superficies tipo Marching Cubes con resultados ópimos. Este tipo de tecnicas tiene problemas en el caso de estar interesado en organos o tejidos blandos, como podrían ser el corazón el higado o los músculos.. debido a que las imágenes correspondientes a estos òrganos suelen presentar contornos muy poco definidos que se confunden con el entorno.

Introducción a Marching Cubes.

Un caso especial de renderización de volúmenes lo constituyen los Marching Cubes. El objetivo de esta técnica es generar el mallado triangular de la superficie externa del objeto 3D contenido en el volumen que se desea renderizar. El mallado se genera localmente teniendo en cuenta la pertenencia o no de los voxels más próximos a uno situado en el límite del objeto. Dicha condición de pertenencia permite la generación de un código de 8 bits a través del cual se puede determinar el mallado a aplicar a dicha zona.

El principio de este algoritmo se basa en que es posible definir un voxel(cubo) por medio de los valores de los pixeles en las ocho esquinas del cubo. Si uno o más pixeles de un cubo tienen valores menores que el isovalor especificado por el usuario y uno o más tienen valores mayores que su valor podemos concluir que voxel deberá contribuir en algun componente de la superficie.

Debido a que las superficies tridimensionales anatómicas ofrecen una herramienta medica muy útil, el algoritmo marching cubes es utilizada para procesar datos medicos. Normalmente, estos datos son adquiridos mediante tomografía computerizada (CT), resonancias magnéticas (MR) o tomografias mediante emisión de fotones simples (SPECT) y el algoritmo marching cubes permite la visualización de modelos complejos.

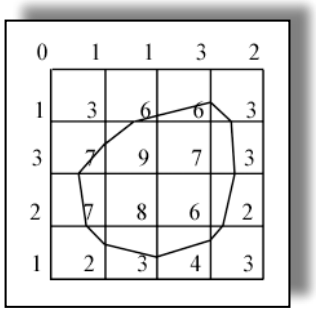


La imagen superior muestra una imagen de la superficie de un cerebro construida mediante el uso de marching cubes.

Marching Squares.

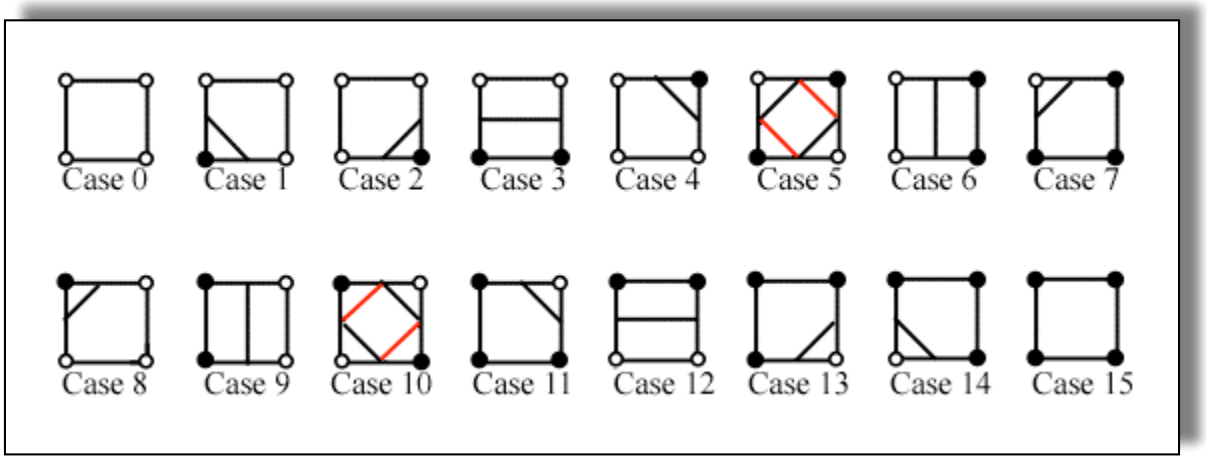
Marching Squares es el algoritmo a partir del cual se han ido generando la mayoría de los algoritmos de reconstrucción 3D y en concreto del que trata este documento, por lo que comprender este algoritmo facilitará su comprensión .Marching Squares permite dibujar líneas entre los valores interpolados a lo largo de los ejes de una superficie, considerando los valores de las esquinas como valores de referencia.

Consideremos una malla 2D como la que se muestra en el siguiente dibujo.

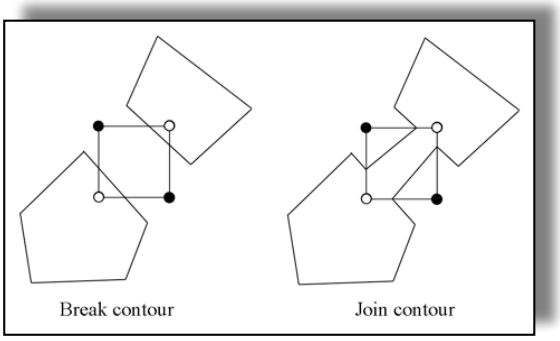


Cada punto de la malla tiene un peso (En la imagen el valor de referencia es de 5) . Para dibujar la curva cuyo valor es constante e igual al de referencia es posible utilizar diferentes clases de interpolacion. El método más utilizado es la interpolación lineal. Con el objetivo de dibujar esta curva, es posible utilizar diferentes metodos de los cuales el principal o mas conocido consiste en considerar individualmente cada superficie de la malla. Este es conocido como el metodo marching square.

Para este método se establecen 16 configuraciones que permiten la representacion de todas las clases de lineas en un espacio bidimensional.



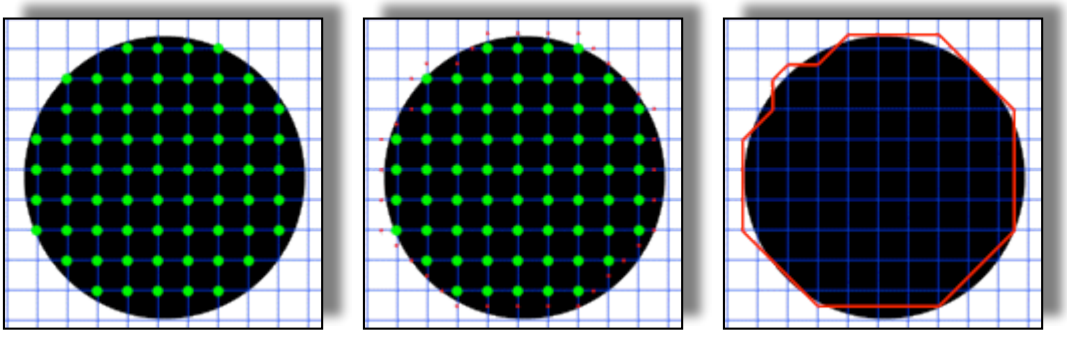
Cuando se analiza este algoritmo en diferentes configuraciones se aprecia que pueden aparecer casos ambiguos como el mostrado en la figura siguiente.



Como se aprecia en la imagen superior no es posible tomar una decision determinista de esta clase de situacion

Ejemplo de construcción de una isosuperficie.

El primer paso consiste en calcular las esquinas que estan dentro de la superficie(reprensentadas mediante puntos verdes). Ahora podemos insertar algunos vertices, de forma que si sabemos que puntos están dentro y que puntos están fuera podemos concluir que un vertice debe estar posicionado aproximadamente a medio camino entre una esquina interior y una esquina exterior que estan conectadas por el eje de una celda. La imagen del centro muestra los vertices interpolados como pequeños puntos rojos mientras que en la imagen de la derecha aparece la superficie formada mediante la unión de los vertices con las líneas.

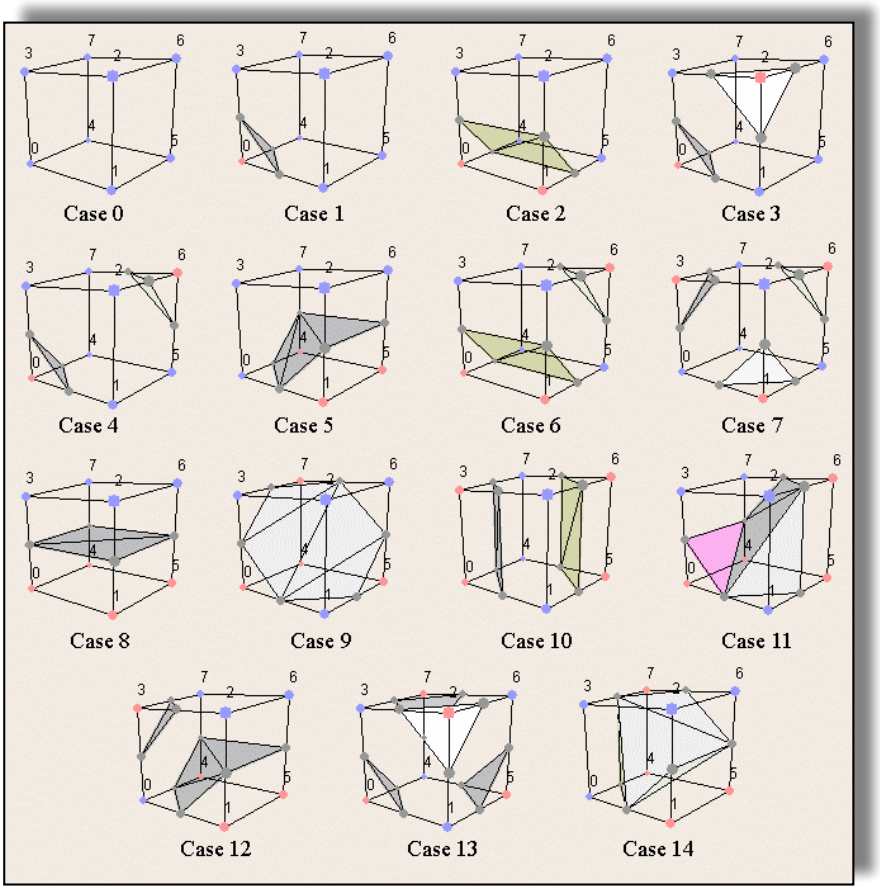


Esquinas dentro de la superficie | Vertices Interpolados | Superficie Reconstruida

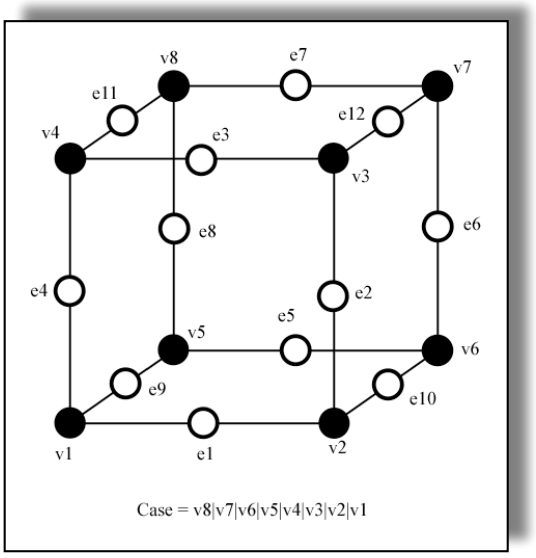
Algoritmo Marching Cubes.

Analizando el algoritmo de los Marching Squares es posible adaptarlo para aprovechar sus funcionalidades en un caso 3D, derivando en el algoritmo marching cubes.

El algoritmo parte de que en un espacio 3D es posible enumerar 256 situaciones diferentes para la representacion de los cubos. Este número de casos puede resultar todavía alto, por lo que mediante rotaciones y simetrias es posible reducir el número a únicamente a las15 familias del dibujo.

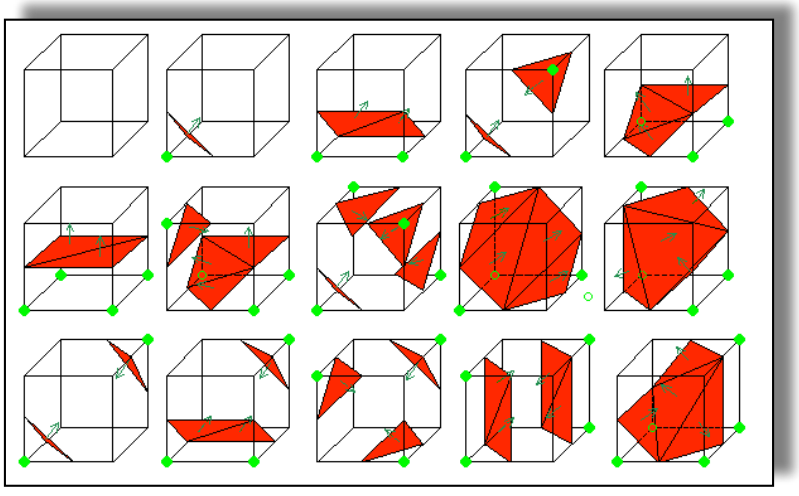


Con el objetivo de poder determinar cada caso real se adopta una dotacion concreta lo que permite referenciar cada caso por un indice creado de la interpretación binaria de los pesos de las esquinas.

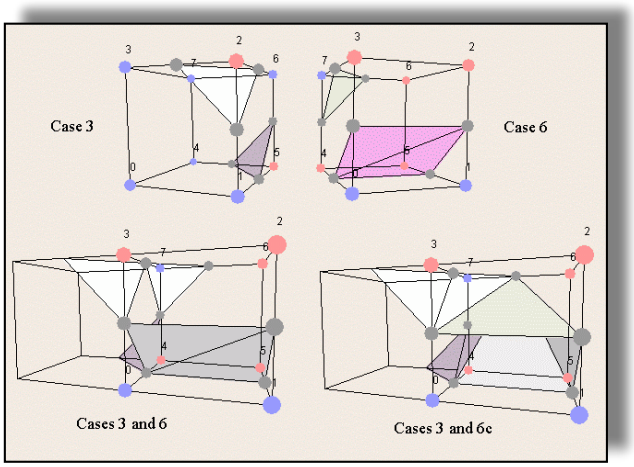


Los vertices del 1 al 8 tienen pesos del 1 al 128 en binario (v1=1, v2=2, v3=4…). De esta forma, un caso en el que son activos los vertices v1 y v3 se corresponde al numero 5 (v1 y v3 son positivos, 1 + 4 = 5).

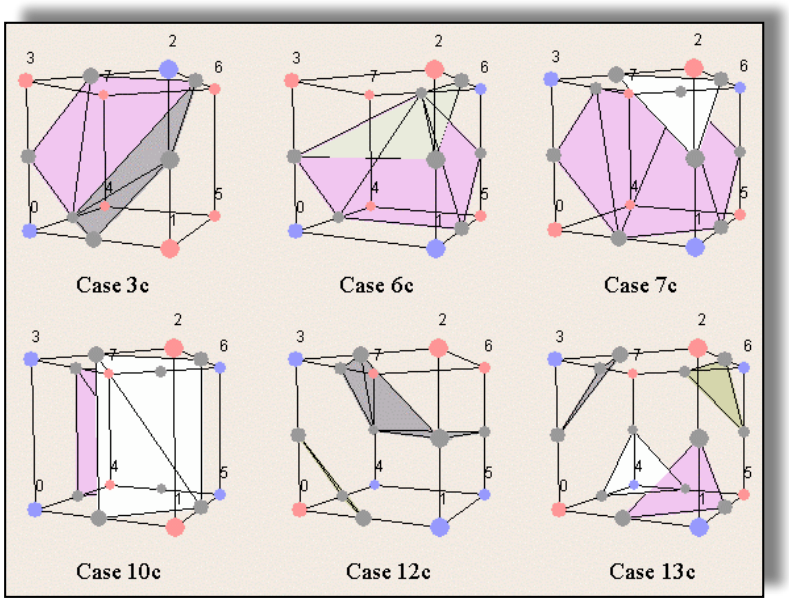
Para todos los cubos mostrados anteriormente exite su complementario. Construir el complementario consiste únicamente en voltear las normales del cubo original. En la siguiente imagen se pueden mostrar algunos ejemplos de cubos asi como sus normales.



Tal y como se explico para el caso de los marching squares en este algoritmo tambien existen algunos casos ambiguos como se mostrará más adelante. En el caso de espacios 3D este inconveniente llega a ser crítico, aunque realmente no son únicamente los casos ambiguos los que pueden crear disfunciones en la topología ya que algunas familias clásicas son incompatibles entre si. La siguiente imagen muestra la problemática de este algoritmo.



La forma más sencilla de solucionar esto y que funciona realmente bien es añadiendo 6 familias nuevas a los casos de marching cubes. Estas familiar deben de ser utilizadas como casos complementarios.



vtkMarchingCubes.

vtkMarchingCubes es un filtro perteneciente a las librerias Vtk desarrolladas por Kitware. Èste toma por entrada un volumen(ej un set de puntos estructurados 3D) y se encarga de generar una salida de una o mas isosuperficies. Es necesario especificar uno o más valores de contorno para generar las isosuperficies,aunque alternativamente, es posible especificar un rango min/max escalar y el numero de contornos para generar una serie de valores de contorno equiespaciados.

vtkMarchingCubes *MCubes=vtkMarchiingCubes::New();
MCubes->SetValue(int i,double valor)

Selecciona un valor de contorno particular en el contorno numero i. El rango de i debe de situarse entre 0 y el Número de Contornos y valor define el valor del
contorno buscado.A su salida tendremos un objeto deltipo PolyData.

Algoritmo de colisiones utilizado

En este proyecto se ha utilizado la librería de detección de colisiones V-Collide.

V-Collide es una librería de detección de colisiones desarrollada por el grupo GAMMA (Geometric Algorithm for modeling, motion and animation por sus siglas en inglés) de la universidad de Carolina del Norte.

La librería está diseñada para operar en ambientes que contienen un gran número de objetos geométricos formados por mallas de triángulos, realizando la detección de colisiones entre parejas de objetos a través de dos etapas:

1. Construir OBB's (hierarchical oriented bounding boxes) para cada objeto, con el fin de encontrar parejas de triángulos que posiblemente intersecten los OBB's.
2. Verificar si las parejas de triángulos detectadas en la etapa 1 realmente se intersectan.

Estas etapas se encuentran en un componente V-Collide llamado RAPID, el cual también es una librería de detección de colisiones independiente. Las diferencias entre ambas librerías son las siguientes:

1. V-Collide conserva información acerca de donde se encuentran los objetos en el ambiente, de tal manera que si éstos no se mueven sus localizaciones no tienen que ser recalculadas, como en RAPID.
2. V-Collide permite la verificación de muchos objetos de forma simultánea.y RAPID solamente permite la verificación de dos.
3. RAPID reporta qué parejas de triángulos exactamente colisionaron, mientras que V-Collide solo reporta colisión entre los objetos.

Algoritmo de construcción del arbol de OBB's.

La construcción del arbol se realiza en dos pasos, primero una colocación de OBB's estrechamente ajustados alrededor de una colección de polígonos y segundo la agrupación de OBB's anidados en una jerarquía en forma de árbol.

En primer lugar, se quiere aproximar la colección de polígonos con un OBB de similares dimensiones y orientación. Se triangularizan los polígonos compuestos de más de tres lados. El algoritmo de cálculo de OBB hace uso de estadísticas de primer y segundo orden resumiendo las coordenadas de los vértices. Se trata de la media, μ, y la matriz de covarianza, C, respectivamente. Si los vértices del triángulo i-ésimo son los puntos p^i, q^i y r^i, entonces la media y la matriz de covarianza pueden ser expresadas en notación vectorial como:

$$\mu = \frac{1}{3n}\sum_{i=0}^{n}\left(p^i + q^i + r^i\right)$$

$$C_{jk} = \frac{1}{3n}\sum_{i=0}^{n}\left(\bar{p}_j{}^i\,\bar{p}_k{}^i + \bar{q}_j{}^i\,\bar{q}_k{}^i + \bar{r}_j{}^i\,\bar{r}_k{}^i\right) \quad 1 \le j,k \le 3$$

Donde n es el número de triángulos, $\bar{p}^i = p^i - \mu$, $\bar{q}^i = q^i - \mu$, $\bar{r}^i = r^i - \mu$. Cada uno de de ellos es un vector 3x1: $\bar{p}^i = (\bar{p}_1^i, \bar{p}_2^i, \bar{p}_3^i)^T$ y C_{jk} son los elementos de la matriz de covarianza 3x3.

Los eigenvectores de una matriz simétrica, como C, son mutuamente ortogonales. Despues de normalizarlos son utilizados como base. Se encuentran los vértices extremos a lo largo de cada eje de esta base. Dos de los tres eigenvectores de la matriz de covarianza son los ejes de la mínima y máxima covarianza, por lo que tenderán a alinear la caja (box) con la geometría de un tubo o de una superficie plana.

El punto débil de la aproximación anterior es que los vertices en el interior del modelo, que seguro no influyen en la selección de la localización del Bounding Boz, pueden tener un impacto arbitrario en los eigenvectores. Por ejemplo, una zona pequeña pero muy densa y planar de vertices en el interior del modelo puede causar la alineación del bounding box con ella. Para mejorar el algoritmo se utiliza el casco convexo de los vértices de los triángulos. El casco convexo es el conjunto convexo más pequeño que contiene todos los puntos y algoritmos eficientes de complejidad O^3(n ln n).

En consecuencia, y para evitar un alineamiento incorrecto, se integra sobre la superficie de cada triangulo de la zona "infinitamente densa" y permitiendo a cada diferencial contribuir a la matriz de covarianza.

---

[3] $O(1)$ hace referencia a tiempo constante. $O(n)$ indica tiempo lineal; $O(n^k)$ (k fijo) indica tiempo polinomial; $O(\log n)$ indica tiempo logarítmico; $O(2^n)$ hace referencia a tiempo exponencial, etc.

Si hacemos que un punto x^i en el triángulo i sea parametrizado por s y t de la siguiente forma:

$$x^i = p^i + s(q^i - p^i) + t(r^i - p^i), \quad s,t \in [0,1]$$

El punto medio del casco convexo es entonces:

$$\mu = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{m^i}\int_0^1\int_0^{1-t} x^i\,ds\,dt\right) = \frac{1}{6n}\sum_{i=1}^{n}\frac{1}{m^i}\left(p^i + q^i + r^i\right)$$

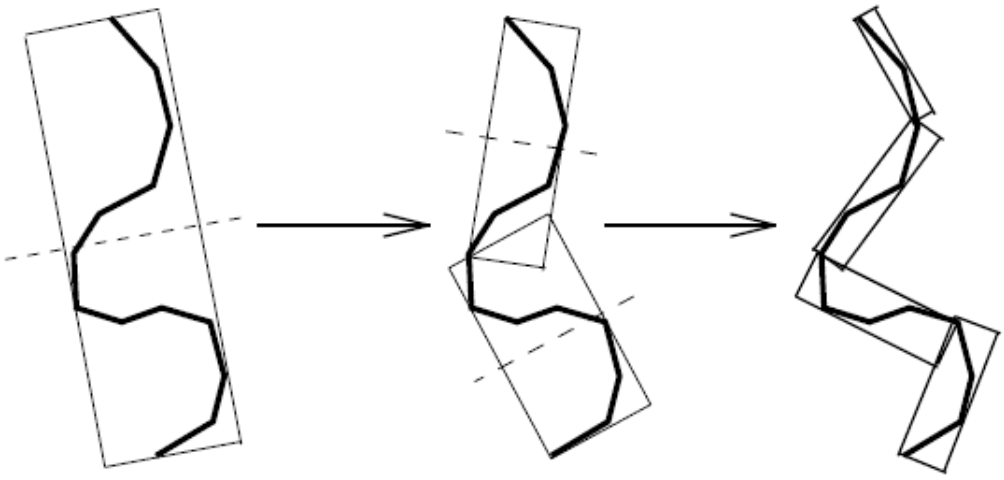
Donde m^i = área del triángulo i-ésimo = $\frac{1}{2}\left|(q^i - p^i)\times(r^i - p^i)\right|$. Los elementos de la matriz de covarianza C tienen la siguiente forma cerrada:

$$C_{jk} = \frac{1}{24n}\sum_{i=1}^{n} m^i\left[\left(\bar{p}_j^{\,i} + \bar{q}_j^{\,i} + \bar{r}_j^{\,i}\right)\left(\bar{p}_k^{\,i} + \bar{q}_k^{\,i} + \bar{r}_k^{\,i}\right) + \bar{p}_j^{\,i}\bar{p}_k^{\,i} + \bar{q}_j^{\,i}\bar{q}_k^{\,i} + \bar{r}_j^{\,i}\bar{r}_k^{\,i}\right], \quad 1 \le j,k \le 3$$

Donde $\bar{p}^i = p^i - \mu$, $\bar{q}^i = q^i - \mu$, $\bar{r}^i = r^i - \mu$.

Dado un algoritmo para calcular OBB's estrechamente ajustados alrededor de un grupo de polígonos ahora necesitamos representarlos jerárquicamente. V-Collide utiliza un método de tipo top-down, es decir, parte de un grupo de todos los polígonos para subdividirlo recursivamente hasta que los nodos hoja sean indivisibles.

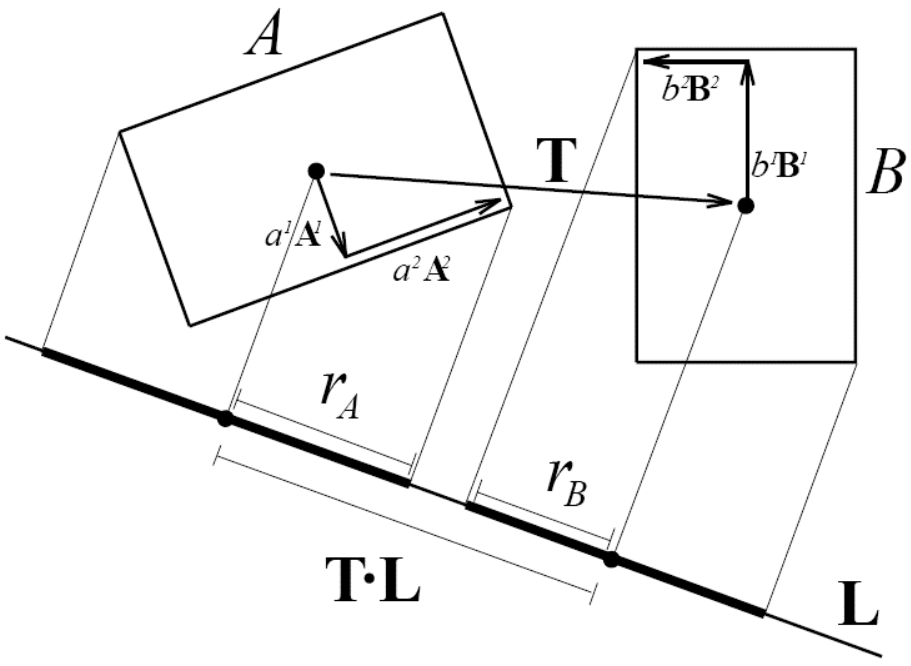
La regla de subdivision que utiliza consiste en dividir el eje más largo de una caja con un plano ortogonal a uno de sus ejes, particionando los polígonos de acuerdo con qué lado del plano reside su punto central. A continuación se muestra un ejemplo análogo en 2D.



Dado un modelo con n triángulos, el tiempo en construir el árbol es $O(n \lg^2 n)$ si utilizamos un casco convexo, y $O(n \lg n)$ en caso contrario.

Dados los OBBTrees de dos objetos, el algoritmo de intersección gasta la mayor parte del tiempo en comprobar si se solapan pares de OBBs. El test que realiza V-Collide consiste en proyecciones axiales de las cajas. Con esta proyección, cada caja forma un intervalo en el eje. Si los intervalos no se solapan entonces las cajas están separadas. Si los intervalos se solapan entonces quizás haya colisión pero es necesario realizar un test más exacto.

¿Cuántos tests son suficientes para determinar el contacto entre dos OBB's? Sabiendo que dos polihedros convexos están separados if existe un eje separador ortogonal a una cara de cualquier polihedro u ortogonal a un borde de cada polihedro. Cada caja tiene 3 únicas orientaciones para los lados y 3 únicas direcciones para los cantos. Esto conlleva a probar 15 ejes potencialmente separadores (3 lados para una caja, 3 lados para la otra caja, 9 combinaciones de pares para los cantos). Si los polihedros están separados, entonces existe algún eje separador, y uno de los quince ejes será un eje separador. Si los polihedros están solapados, entonces no existirá ningún eje separador. Por eso, probar los 15 ejes es un test suficiente para determinar si se solapan dos OBB's.



En la figura anterior se observa (en 2D) como L es un eje separador de los OBB's A y B ya que las proyecciones de A y B en L se corresponden con intervalos separados.

Para realizar el test, la estrategia empleada por V-Collide es proyectar los centros de las cajas en el eje y tambien calcular el radio de los intervalos. Si la distancia entre los centros de las cajas proyectados en el eje es más grande que la suma de los radios, entonces los intervalos (y por tanto, las cajas) están separados.

Asumiendo dos OBB's, A y B, con B colocado una posición relativa a A mediante una rotación $\vec{R}$ y una traslación $\vec{T}$. Siendo a_i y b_i los radios de A y B donde i=1,2,3. Llamando a los ejes de A y B como los vectores unitarios $\overline{A}^i$ y $\overline{B}^i$, para i=1,2,3, hablaremos entonces de los 6 ejes de las cajas. Nótese que si utilizamos los ejes de la caja de A como base, las tres columnas de R son las mismas que los tres vectores $\overline{B}^i$. Los centros de cada caja se proyectan en el punto medio del intervalo. Proyectando los radios de la caja en el eje, y sumando la longitud de sus imágenes, se obtiene el radio del intervalo. Si el eje es paralelo al vector unitario $\vec{L}$ entonces el radio del intervalo de la caja A es:

$$r_A = \sum_i \left| a_i \vec{A}^i \cdot L^i \right|$$

De forma similar se obtiene la expresión para r_B.
La colocación del eje es inmaterial por lo que asumimos que pasa a través del centro de la caja A. La distancia entre los puntos medios de los intervalos son intervalos $\left|\vec{T} \cdot \vec{L}\right|$.
Por tanto, los intervalos están separados si se cumple:

$$\left|\vec{T} \cdot \vec{L}\right| > \sum_i \left| a_i \vec{A}^i \cdot L^i \right| + \sum_i \left| b_i \vec{B}^i \cdot L^i \right|$$

No obstante, y dependiendo de las condiciones (por ejemplo si L se puede obtener como producto vectorial de A y B, etc..) esta expresión puede simplificarse.

Con que se cumpla un única expresión, cualquiera de las 15 restantes pruebas es innecesaria. En el peor de los casos se requieren aproximadamente 200 operaciones pero normalmente se necesitarán muchas menos.

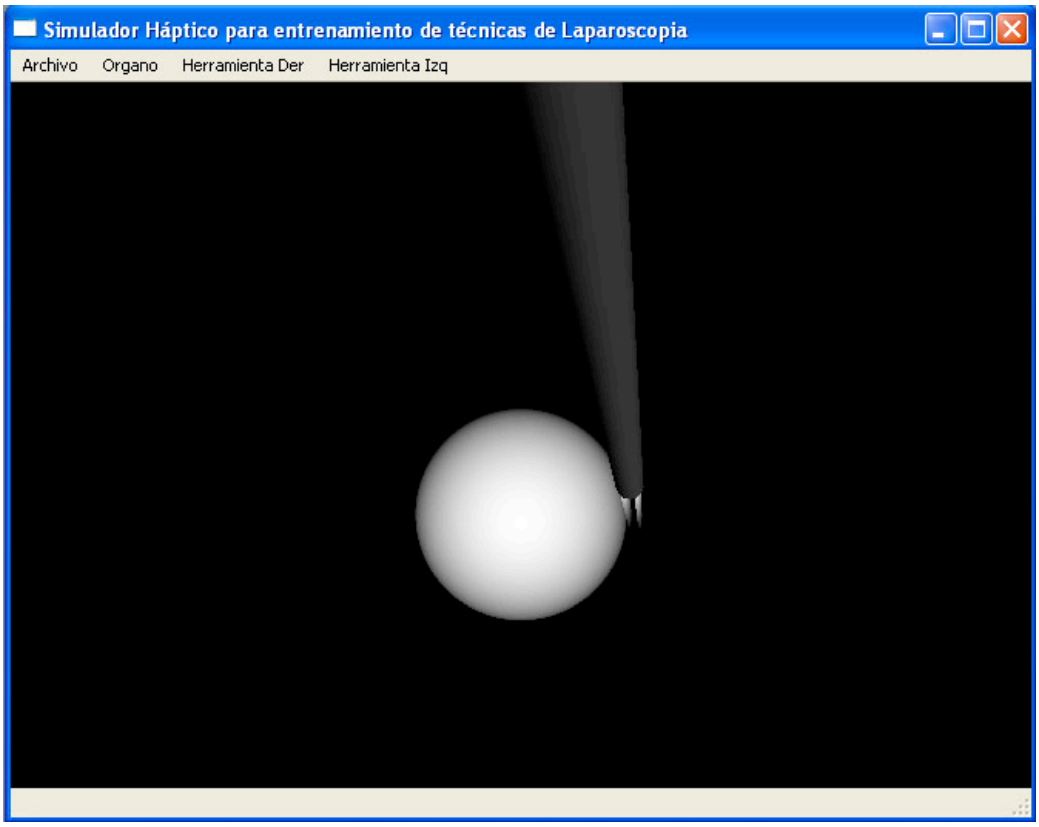
Capítulo 4. Detección de colisiones.

El simulador de laparoscopia realizado implementa un sistema de detección de colisiones basado en las librerías V-Collide de tal forma que se detecta colisión entre:

- Órgano-Herramienta
- Herramienta-Herramienta

Realmente el sistema de colisiones trabaja a nivel de actor[4] por lo que por ejemplo, se podría configurar para que determinadas partes de una herramienta no fueran colisionables.

En la siguiente imagen se puede observar una instantánea del momento en el que se produce colisión entre una esfera y una herramienta de laparoscopia.



La integración de un sistema de colisiones en este proyecto de Laparoscopia ha supuesto un importante desafío dada la escasa, prácticamente nula, información encontrada en Internet sobre VTK y detección de colisiones.

[4] Actor hace referencia al concepto de actor en VTK.

Introducción a las librerías V-Collide

V-Collide es una librería de detección de colisiones completamente gratuita para uso no comercial desarrollada por el grupo GAMMA (Geometric algorithm for modeling, motion and animation) de la universidad de Carolina del Norte (The University of North Carolina at Chapel Hill). Esta librería realiza una detección rápida y exacta de colisiones entre modelos poligonales triangulados, por lo que puede ser usada para realidad virtual y en una gran variedad de aplicaciones de simulación. La librería esta diseñada para operar en ambientes que contienen un gran número de objetos geométricos formados con mallas de triángulos.

La última versión disponible de V-Collide es la 2.01. El código fuente está escrito en C++ y es descargable desde la siguiente dirección de Internet:

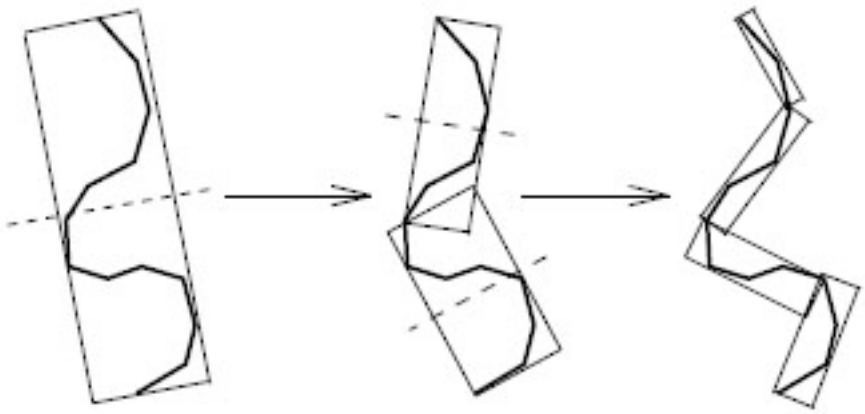
http://www.cs.unc.edu/~geom/V_COLLIDE/request.html

Los pasos básicos para el uso de esta librería son: crear los objetos, añadir conjuntos de triángulos a estos objetos, elegir que pares de objetos deben ser evaluados, cambiar la posición de estos objetos actualizando la matriz, ejecutar el test de colisiones y conseguir los resultados del test. Los tres últimos pasos se repiten para cada paso de la simulación.

V-Collide utiliza una arquitectura de detección de colisiones en tres etapas:

- Un test N-body encuentra pares de objetos posiblemente colisionados.
- Un test jerárquico de OBBs (Oriented Bounding Boxes) encuentra posibles pares de triángulos colisionados.
- Un test exacto determina si un par de triángulos están solapados.

La rutina N-body tiene coherencia entre sucesivos intervalos de tiempo de una simulación para tener un buen rendimiento en simulaciones de animaciones y movimiento. Las jerarquías de OBBs (Oriented Bounding Boxes) y los algoritmos exactos de colisión se toman de RAPID, un componente de V-Collide que también está disponible como un paquete independiente.



Integración de VTK y Vcollide

Para integrar VTK y VCollide en el proyecto de Laparoscopia se ha desarrollado una clase intermediaria entre VTK y VCollide: CollisionObjectFactory.

En la clase CollisionObjectFactory, de ahora en adelante COF, se ha desarrollado un método fundamental para añadir un objeto VTK al sistema VCollide, el método AddObject.

Este método se encarga de hacer el siguiente procesado con el objeto que se le pasa como parámetro:

- triangularización de los polígonos
- escalado en los ejes X,Y y Z
- obtención de información de los triángulos (coordenadas de los vértices y asociaciones de vértices) y almacenamiento en matrices dinámicas.
- creación de un objeto Vcollide utilizando para ello la información de los triángulos almacenada en las matrices dinámicas.

La definición del método AddObject:

```
// Añade objeto al sistema VCollide y almacena el indice del array
PiezasArray
      void AddObject(int* id,vtkAlgorithm *algorithm, double
scaleX,double scaleY,double scaleZ,char* nombre, vtkActor* actor, int*
PA_indice);
```

Veamos a continuación qué son y para que sirven cada uno de los parámetros de este método AddObject.

El parámetro id. Identificador de Vcollide. Vcollide utiliza un identificador para cada objeto añadido al sistema de colisiones. Necesita que se le pase un puntero a entero donde almacenar el identificador que le corresponda. Se trata de una asignación ascendente, empieza en cero y no se resetea nunca.

El parámetro algorithm. Se trata del objeto que queremos añadir al sistema de colisiones y puede ser cualquier tipo de objeto fuente disponible en vtk como por ejemplo un vtkSphereSource, vtkConeSource, etc.. así como un vtkOBJReader, si hemos obtenido el objeto a partir de un archivo .obj (utilizado por ejemplo para obtener las piezas que componen las herramientas de laparoscopia).

Los parámetros scaleX, scaleY y scaleZ.

Para el que ya conozca VCollide, sabrá que tiene ciertas limitaciones o requisitos como por ejemplo que no soporta escalado de objetos y que requiere que los polígonos de los objetos sean triángulos. Estas limitaciones no hay que tenerlas en cuenta ya que AddObject, previamente a su interacción con VCollide, incluye un filtro de triangularización y un filtro de escalado (parámetros scaleX,scaleY,scaleZ).

Los parámetros scaleX, scaleY y scaleZ sirven para escalar el objeto en los ejes X,Y y Z respectivamente.

Los posibles valores que pueden tomar estos parámetros de escalado son:

- 1, indica que no hay escalado
- 0<x<1, indica una reducción del tamaño
- >1, indica que un aumento del tamaño

El parámetro nombre. Sirve para asignar un nombre a la pieza colisionable.

El parámetro actor. El actor que representa a la pieza colisionable.

El parámetro PA_indice. Índice del array PiezaArray. Para llevar el control de los objetos colisionables que se están utilizando en la aplicación se ha creado un Array dinámico de tamaño `TAM_MAX_PIEZAS_ARRAY`: PiezasArray, de ahí el primer término de este parámetro: PA. De igual forma que el parámetro id, el valor del índice se asignará desde el propio método AddObject.

```
	// Array con información sobre las piezas añadidas al sistema de
colisiones
	ObjectInfo* PiezasArray;

// Structura de datos sobre cada objeto/pieza del sistema de
colisiones
	struct ObjectInfo {
		int VC_Id; // ID en VCollide
		char nombre[50]; // Nombre de la pieza
		vtkActor* actor; //actor correspondiente a la pieza
		vtkTransformPolyDataFilter* filtro; //filtro
(triangularizado y escalado)
		bool isOrgano; //diferencia organos y herramientas
	};
```

Por último, se procede como es habitual en VTK, es decir, creación de un mapper, asociación del mapper al actor y por último se añade el actor al Render.

Es muy importante tener en cuenta que el mapper necesita el filtro del objeto triangularizado y escalado. Para obtenerlo basta utilizar el método `GetFilterOfPiezasArray` de la clase CollisionObjectFactory pasándole como parámetro el indice del array PiezasArray.

Ejemplo de creación de un objeto colisionable.

Veamos un ejemplo de cómo crear un objeto colisionable, en este caso un cono:

```
// Creacion un objeto CollisionObjectFactory
CollisionObjectFactory COF = new CollisionObjectFactory();

//creacion de un cono en vtk
vtkSphereSource *cone = vtkSphereSource::New();
cone->SetRadius(50.0);
cone->SetPhiResolution(100);
cone->SetThetaResolution(100);

vtkActor *coneActor;
int PA_id;
int id;
```

```
// añadir objeto al sistema de colisiones
COF->AddObject(&id, cone,1,1,1,"Organo",coneActor,&PA_id);

vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();

//Si el objeto no fuera colisionable: cone->GetOutputPort()
//coneMapper->SetInputConnection( cone->GetOutputPort() );
//

// Para objeto colisionable obtenemos el filtro de COF
coneMapper->SetInputConnection(COF->GetFilterOfPiezasArray(PA_id)-
>GetOutputPort());

coneActor = vtkActor::New();
coneActor->SetMapper( coneMapper );
coneActor->GetProperty()->SetDiffuseColor(0.6,0.6,0.1);
ren->AddActor( coneActor );
```

Las diferencias respecto a crear un objeto “no colisionable” son las siguientes líneas:

```
COF->AddObject(&this->id, cone,1,1,1,"Organo",coneActor,&PA_id);

coneMapper->SetInputConnection(COF->GetFilterOfPiezasArray(PA_id)-
>GetOutputPort());
```

donde id y PA_id son variables tipo Int que sirven para identificar al objeto en VCollide y en el sistema de colisiones (CollisionObjectFactory) respectivamente. Estas variables no hay que inicializarlas ya que es dentro del propio método AddObject desde donde se les asignará valor.

El método AddObject se encarga de procesar el objeto (triangularizar y escalar antes de añadir al sistema de colisiones) y de guardar el filtro con el procesamiento realizado.

```
void AddObject(int* id,vtkAlgorithm *algorithm, double scaleX,double
scaleY,double scaleZ,char* nombre, vtkActor* actor, int* PA_indice);
```

Para recuperar el filtro utilizaremos el método GetFilterOfPiezasArray:

```
vtkTransformPolyDataFilter* GetFilterOfPiezasArray(int PA_index) {
return this->PiezasArray[PA_index].filtro; }
```

El método AddObject:

```
void CollisionObjectFactory::AddObject(int *id,vtkAlgorithm
*algorithm, double scaleX, double scaleY, double scaleZ, char* nombre,
vtkActor* actor, int* PA_indice)
{
	// Para añadir un objeto al sistema de colisiones VCollide hay
	// que tener en cuenta dos consideraciones:
	// 1) Triangularizar
	// 2) No soporta escalado, por lo que el escalado debe hacerse
	//    a priori utilizando vtkTransformPolyDataFilter

	//std::cout << "Añadiendo objeto al sistema de colisiones\n";
	// Triangularizamos
	vtkTriangleFilter* triObject = vtkTriangleFilter::New();
	triObject->SetInputConnection(algorithm->GetOutputPort());
	triObject->SetPassLines(5);
```

```
	// Escalamos
	vtkTransform *tr = vtkTransform::New();
	tr->Scale(scaleX,scaleY,scaleZ);

	vtkTransformPolyDataFilter *algorithmFiltered =
vtkTransformPolyDataFilter::New();
	algorithmFiltered->SetInputConnection(triObject-
>GetOutputPort());
	algorithmFiltered->SetTransform(tr);

	// para que getpoints no devuelva cero
	triObject->Update();
	algorithmFiltered->Update();

	vtkPoints *puntos = vtkPoints::New();
	puntos=algorithmFiltered->GetOutput()->GetPoints();

	//std::cout << "matriz dinámica de " << puntos-
>GetNumberOfPoints() << " puntos" << std::endl;

	// creamos matriz dinamica para almacenar coordenadas de los
puntos
	double **matriz;
	matriz = new double* [puntos->GetNumberOfPoints()];

	for (int i=0;i<puntos->GetNumberOfPoints();i++)
	  {
	  matriz[i]=new double[3];
	  }

	// almacena las coordenadas de los puntos en la matriz dinámica
	for (int i=0;i<puntos->GetNumberOfPoints();i++)
	  {
		puntos->GetPoint(i,matriz[i]);
	  }

	// crea un nuevo objeto VCollide y lo prepara para añadirle
triangulos
	vc.NewObject(id);

	//Despues de llamar a NewObject VCollide ya ha asignado un id
	int numObj=*id;

	cout << "COF ha asignado para un nuevo objeto el id=" <<
numObj<<"\n";

	// Almacena información sobre el objeto en PiezasArray
	*PA_indice = this->SetDataOfObjectInPiezasArray(numObj, nombre,
actor, algorithmFiltered);

	cout << "COF ha asignado para un nuevo objeto el id interno=" <<
*PA_indice<<"\n";

	//añadimos al vector de objetos un puntero al objeto ya filtrado
(escalado y triangularizado)
	//this->objetos[numObj]=algorithmFiltered;

	vtkCellArray *celdas = vtkCellArray::New();
	celdas=algorithmFiltered->GetOutput()->GetPolys();
```

```
	//identificador de triangulo
	int idTri=0;

	vtkIdType npts;
	vtkIdType *pts;

	//añade los triangulos en base a la informacion de las celdas
	while (celdas->GetNextCell(npts,pts)!=0)
	  {
	  vc.AddTri(matriz[pts[0]],matriz[pts[1]],matriz[pts[2]],idTri);
	  idTri++;
	  }

	vc.EndObject();

	//std::cout << "Objeto añadido OK al sistema de colisiones\n";

	// se destruye la matriz dinamica
	for (int i=0;i<puntos->GetNumberOfPoints();i++)
	      {
	      delete [] matriz[i];
	      }
	delete [] matriz;

	//std::cout << "Matriz dinamica destruida\n";


	tr->Delete();
	//algorithmFiltered->Delete();	//celdas y puntos no hace
falta
	triObject->Delete();
	cout << "Fin de AddObject\n";
}
```

El método `SetDataOfObjectInPiezasArray`

```
// rellena un elemento del array PiezasArray y devuelve el indice en
el array o -1 (error)
	int SetDataOfObjectInPiezasArray(int VC_Id, char* nombre,
vtkActor* actor, vtkTransformPolyDataFilter* filtro);
```

La clase CollisionObjectFactory

La clase CollisionObjectFactory provee de diferentes métodos para el tratamiento de las colisiones así como de la estructura de datos para almacenar información sobre los objetos añadidos al sistema de colisiones.

Las funcionalidades más importantes de esta clase son:

- añadir objetos al sistema de colisiones
- eliminar objetos del sistema de colisiones
- desactivar la colisión entre pares de objetos
- actualizar la matriz de transformación de un ensamblado
- determinar qué órgano ha colisionado con alguna herramienta
- determinar si una determinada pieza ha colisionado
- calcular colisiones
- informar sobre colisiones

Añadir objetos al sistema de colisiones. El método AddObject.

Este método se encarga de hacer el siguiente procesado con el objeto que se le pasa como parámetro:

- triangularización de los polígonos
- escalado en los ejes X,Y y Z
- obtención de información de los triángulos (coordenadas de los vértices y asociaciones de vértices) y almacenamiento en matrices dinámicas.
- creación de un objeto Vcollide utilizando para ello la información de los triangulos almacenada en las matrices dinámicas.
- Almacenar información sobre el objeto en la estructura de datos PiezasArray

La definición del método AddObject:

```
// Añade objeto al sistema VCollide y almacena el indice del array
PiezasArray
      void AddObject(int* id,vtkAlgorithm *algorithm, double
scaleX,double scaleY,double scaleZ,char* nombre, vtkActor* actor, int*
PA_indice);
```

Para un ejemplo sencillo de funcionamiento de este método así como una explicación de los parámetros ver el punto de Integración de VTK y Vcollide.

Para eliminar un objeto del sistema de colisiones se utiliza el método DeleteObject, pasándole de parámetro el PA_index, es decir, el identificador del objeto en PiezasArray.

Desactivar la colisión entre un par de objetos.

En ocasiones se desea que un objeto colisionable no sea colisionable con algún otro objeto. Por ejemplo, si disponemos de una herramienta Lapiz que se componga de dos piezas: el palo y la punta. Podemos desear que tanto el palo como la punta sean objetos

colisionables y que se detecte su colisión con otras herramientas o con órganos pero nunca entre ellos.

Para desactivar la colisión entre pares de objetos se han desarrollado dos métodos: `DeactivateCollision y DeactivateCollisionWithAnotherOrganos.`

El método DeactivateCollision sirve para desactivar la colisión entre dos piezas a partir de los nombres con los que han sido añadidos al sistema de colisiones. Por ejemplo, si hemos añadido una herramienta lápiz que se compone de dos objetos en el sistema de colisiones cuyos nombres son palo y punta respectivamente, para desactivar la colisión entre ambos:

```
char* objectsNameArray[] = {"palo","punta"};
// Desactiva colisiones
this->OCF-
>DeactivateCollision(objectsNameArray[0],objectsNameArray[1]);
```

Este método recorre PiezasArray para obtener los identificadores de ambas piezas en Vcollide y desactivar su colisión mediante el método `DeactivatePair` de la librería Vcollide.

Por otro lado, el método `DeactivateCollisionWithAnotherOrganos` sirve para desactivar la colisión entre el órgano cuyo índice en PiezasArray se pasa como parámetro y el resto de órganos.

Este otro método recorre PiezasArray en busca de órganos y desactiva la colisión entre todos los que encuentre.

Actualizar la matriz de un ensamblado.

Las herramientas de laparoscopia se componen de una o más piezas. Un ensamblado es una agrupación de una o más piezas por lo que las herramientas estarán formadas mínimo por un ensamblado. Si se trata de una herramienta articulada tendrá que estar formada por dos ensamblados, un ensamblado principal y otro ensamblado para la pieza articulada.

Cada objeto en VTK tiene asociada una matriz 4x4 de transformación que representa de forma matemática las transformaciones asociadas en los tres ejes: rotación, traslación y escalado. Realmente, Vcollide no soporta escalado, el escalado se realiza en el método AddObject, antes de añadir el objeto al sistema de colisiones, por lo que las transformaciones a las que haremos referencia a partir de ahora serán rotación y traslación.

Cuando realicemos una operación de movimiento, ya sea de tipo rotacional o traslacional, sobre un objeto 3D colisionable, tenemos que actualizar las matrices de transformación de los objetos/ensamblados que componen dicho objeto.

Para actualizar las matrices de transformación de los actores que componen una herramienta, la clase CollisionObjectFactory dispone de dos métodos: `UpdateAssemblyMatrix y UpdateAssembliesMatrices`, para herramientas que se

compongan de uno o de dos ensamblados respectivamente. Como el primer método se podría considerar una versión mucho más simplificada del método para más de un ensamblado, pasaremos a hablar de este último.

En el caso de que la herramienta se componga de dos ensamblados se presupone la siguiente hipótesis:

- la herramienta se compone de un total de x actores
- el ensamblado principal se compone de un actor y de un ensamblado secundario
- el ensamblado secundario se compone de x-1 actores

Mediante operaciones de concatenación de matrices, el método `UpdateAssembliesMatrices`, obtiene los valores de la matriz con los que tendrá que actualizar su correspondiente matriz en Vcollide.

De esta forma, resulta muy sencillo actualizar la matriz de transformación de una herramienta de dos ensamblados:

```
void CollisionObjectFactory::UpdateAssembliesMatrices(vtkAssembly*
ensamblado, vtkAssembly* ensamblado2, int actorsIdArray[],
vtkTransform* transformArray[])
```

Ejemplo de actualización de una herramienta de laparoscopia:

```
// actualiza la matriz del ensamblado
      this->OCF->UpdateAssembliesMatrices(this->Ensamblado, this-
>Ensamblado2, this->ActorsVCIdArray, this->TransformArray);
```

Como se puede observar en la definición del método, requiere que la herramienta tenga asociada un array de identificadores en Vcollide así como un array de objetos vtkTransform para el cálculo de las correspondientes matrices de transformación.

Determinar qué órgano ha colisionado con alguna herramienta

En el caso de que haya habido colisión entre un par de objetos quizás nos interese determinar si ha habido colisión de un órgano con una herramienta, para ello podemos hacer uso del método `GetOrganoCollisionedWithHerramienta` :

```
// devuelve el Id del organo que ha colisionado con la herramienta. Si
devuelve -1 no hay colision organo-herramienta
      int GetOrganoCollisionedWithHerramienta(int Herr_VC_id);
```

En el caso de que haya colisión de la herramienta con algún órgano, devuelve el identificador en Vcollide del órgano.

Determinar si una pieza ha colisionado

HasCollisioned permite saber si una determinada pieza colisionable ha colisionado.

```
// devuelve 1 si hay colision
int HasCollisioned(int VC_id);
```

Por último, para calcular colisiones y obtener un informe de colisiones podemos utilizar los métodos `CalculateCollisions() e InformAboutCollisions()`. Este último método, antes de mostrar información sobre las colisiones llama al método `CalculateCollisions()`.

```
int CollisionObjectFactory::HasCollisioned(int VC_id)
{
	int j;
	for (j = 0; j < report.numObjPairs(); j++)
		if ((report.obj1ID(j)==VC_id || report.obj2ID(j)==VC_id))
		  {
			//std::cout << "Q" << VC_id << " HasCollisioned::: 
"<<report.obj1ID(j)<<" y "<<report.obj2ID(j)<<std::endl;
			return 1;
		  }
	return 0;
}
```

Por último, destacar que V-Collide no entiende de distancias, simplemente detecta si hay o no colisión.

En el caso de colision, para saber qué triángulos han colisionado es muy importante el buen conocimiento de la clase VCReport de VCollide.

La clase VCReport consta de los siguientes métodos:

`int numObjPairs();`	Número de pares de objetos en contacto
`int obj1ID( int obj_pair_num );`	ID del primer objeto especificado en el par de objetos
`int obj2ID( int obj_pair_num );`	ID del segundo objeto especificado en el par de objetos
`int numTriPairs( int obj_pair_num );`	Devuelve el número de pares de triángulos que están en contacto entre los pares de objetos especificados. (útil sólo cuando se llama a Vcollide::Collide() con VC_ALL_CONTACTS.)
`int tri1ID( int obj_pair_num, int tri_pair_num );`	ID del primer triángulo en el par de triángulos específicado. Tri_pair_num va desde 0 hasta el número de pares de triángulos – 1
`int tri2ID( int obj_pair_num, int tri_pair_num );`	ID del segundo triángulo en el par de triángulos específicado. Tri_pair_num va desde 0 hasta el número de pares de triángulos – 1

A modo de ejemplo, el siguiente código se utiliza por la herramienta Cutter para obtener qué triángulos tiene que colorear:

```
void Cutter::clicIzq()
{
	this->OCF->InformAboutCollisions();
	// obtiene el id del organo colisionado
	int organo_PA_Id=this->OCF-
>GetOrganoCollisionedWithHerramienta(this->ActorsVCIdArray[1]);
```

```
      // comprueba si ha colisionado el objeto invisible y si ademas
ha colisionado con un organo
      if (this->OCF->HasCollisioned(this-
>ActorsVCIdArray[1])&&organo_PA_Id!=-1)
        {
            //tiene que pintar
            int numCeldas;
            scalars=vtkUnsignedCharArray::SafeDownCast(this->OCF-
>GetFilterOfPiezasArray(organo_PA_Id)->GetOutput()->GetCellData()-
>GetScalars());
            if (scalars==NULL)
                  {
                  numCeldas=this->OCF-
>GetFilterOfPiezasArray(organo_PA_Id)->GetOutput()-
>GetNumberOfCells();
                  std::cout << "generando scalars..para numceldas=" <<
numCeldas << std::endl;
                  scalars=vtkUnsignedCharArray::New();
                  scalars->SetNumberOfComponents(4);
                  scalars->SetNumberOfTuples(numCeldas);
                  float r=0;float g=0;float b=0;
                  for (int i=0;i<numCeldas;i++)
                        {
                        scalars->SetTuple4(i,255,255,255,255);
                        }
                  this->OCF->GetFilterOfPiezasArray(organo_PA_Id)-
>GetOutput()->GetCellData()->SetScalars(scalars);

                  }
            this->OCF->GetFilterOfPiezasArray(organo_PA_Id)-
>GetOutput()->GetCellData()->Modified();
            //PINTA
            int obj_pair_num=this->OCF-
>GetCollisionedPairOfObjectsNum(this->ActorsVCIdArray[1]);
            int num_tri;
            // para saber si se ha de colorear tri1ID o tri2ID
            int obj_id=this->OCF->report.obj1ID(obj_pair_num);
            bool tri1=false;
            if (obj_id==organo_PA_Id) tri1=true;
            else tri1=false;

            for (int k=0;k<this->OCF-
>report.numTriPairs(obj_pair_num);k++)
                  {
                        if (tri1==true) {
                              num_tri=this->OCF-
>report.tri1ID(obj_pair_num,k);
                        }
                        else {
                              num_tri=this->OCF-
>report.tri2ID(obj_pair_num,k);
                        }

                        }// pinta de color rojo
                        scalars->SetTuple4(num_tri,220,17,17,255);
                        }
            this->OCF->GetFilterOfPiezasArray(organo_PA_Id)-
>GetOutput()->GetCellData()->Modified();
            }
      else std::cout << "No puede pintar.\n";
```

Capítulo 5. Creación de herramientas 3D con VTK

Introducción

En este capítulo se explica el proceso de creación de una herramienta 3D para utilizar desde el simulador de Laparoscopia.

Los pasos a seguir son:

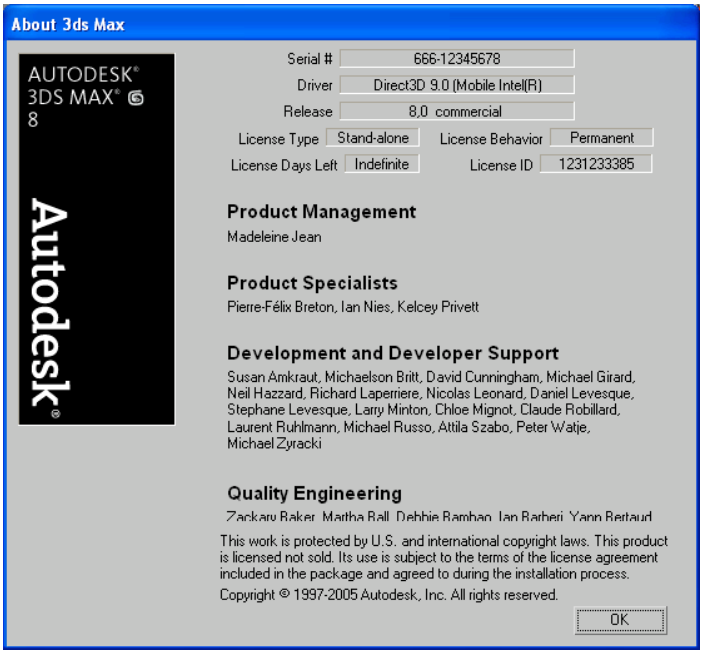
- Diseño de las piezas que componen la herramienta
- Almacenado de las piezas en un formato adecuado
- Desarrollo de la clase en base a alguna de las existentes

Para el primer paso se requiere el uso de un software de diseño gráfico en 3D que permita la exportación del diseño a un formato gráfico soportado por VTK. Para el caso que nos ocupa utilizaremos el software 3DS Max, que permite guardar los objetos 3D diseñados en formato wavefront (archivos con extensión .obj), formato fácil de importar desde VTK, pero se podría haber utilizado cualquier otro software.

Diseño de herramientas con 3DS Max.

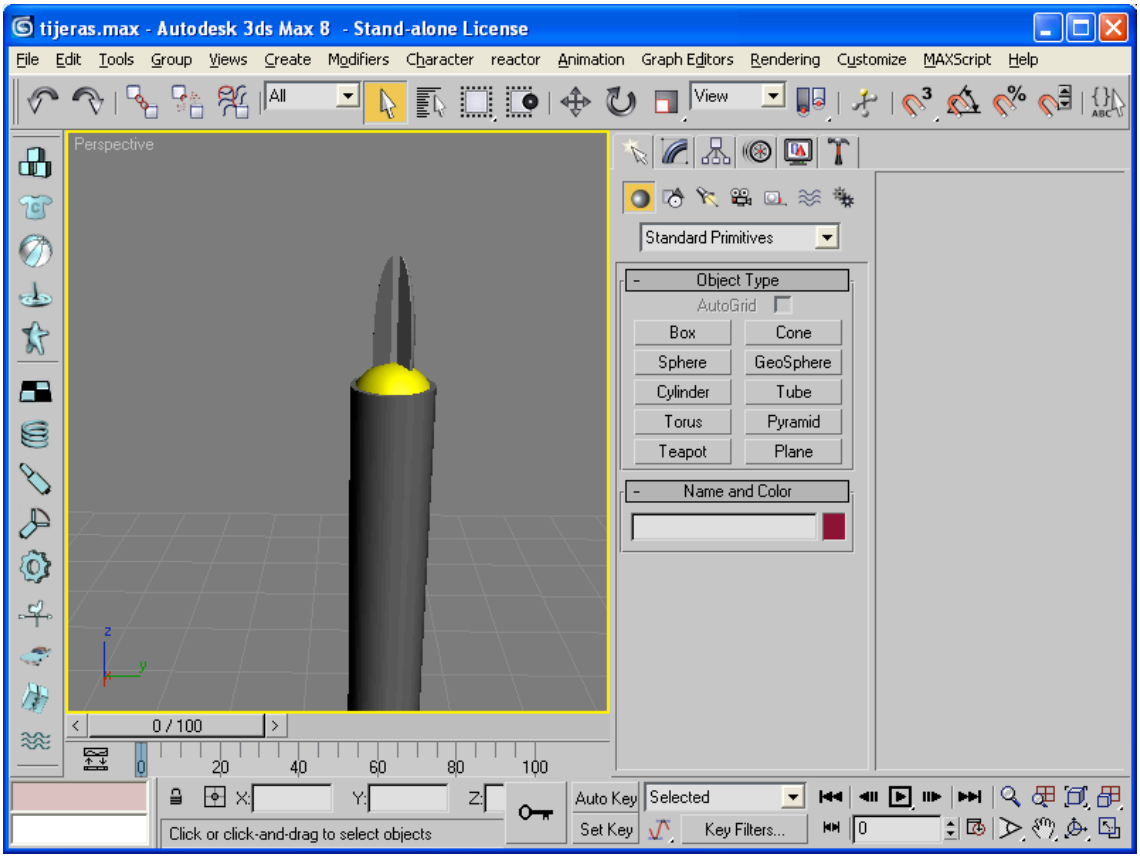
Como se ha comentado en el punto anterior se podría haber utilizado cualquier otro editor gráfico 3D pero se ha optado por 3DS Max por tratarse de un software ampliamente utilizado y universalmente conocido.

La versión utilizada de este software ha sido la versión 8.


About 3ds Max
AUTODESK
3DS MAX
8
Autodesk
Serial # 666-12345678
Driver Direct3D 9.0 (Mobile Intel(R)
Release 8.0 commercial
License Type Stand-alone
License Behavior Permanent
License Days Left Indefinite
License ID 1231233385
Product Management
Madeleine Jean
Product Specialists
Pierre-Félix Breton, Ian Nies, Kelcey Privett
Development and Developer Support
Susan Amkraut, Michaelson Britt, David Cunningham, Michael Girard, Neil Hazzard, Richard Laperriere, Nicolas Leonard, Daniel Levesque, Stephane Levesque, Larry Minton, Chloe Mignot, Claude Robillard, Laurent Ruhlmann, Michael Russo, Attila Szabo, Peter Watje, Michael Zyracki
Quality Engineering
This work is protected by U.S. and international copyright laws. This product is licensed not sold. Its use is subject to the terms of the license agreement included in the package and agreed to during the installation process.
Copyright © 1997-2005 Autodesk, Inc. All rights reserved.
OK

En este apartado se explica a modo de ejemplo cómo diseñar una herramienta de tipo tijeras o pinzas.

El diseño final de la herramienta será el que se muestra a continuación:



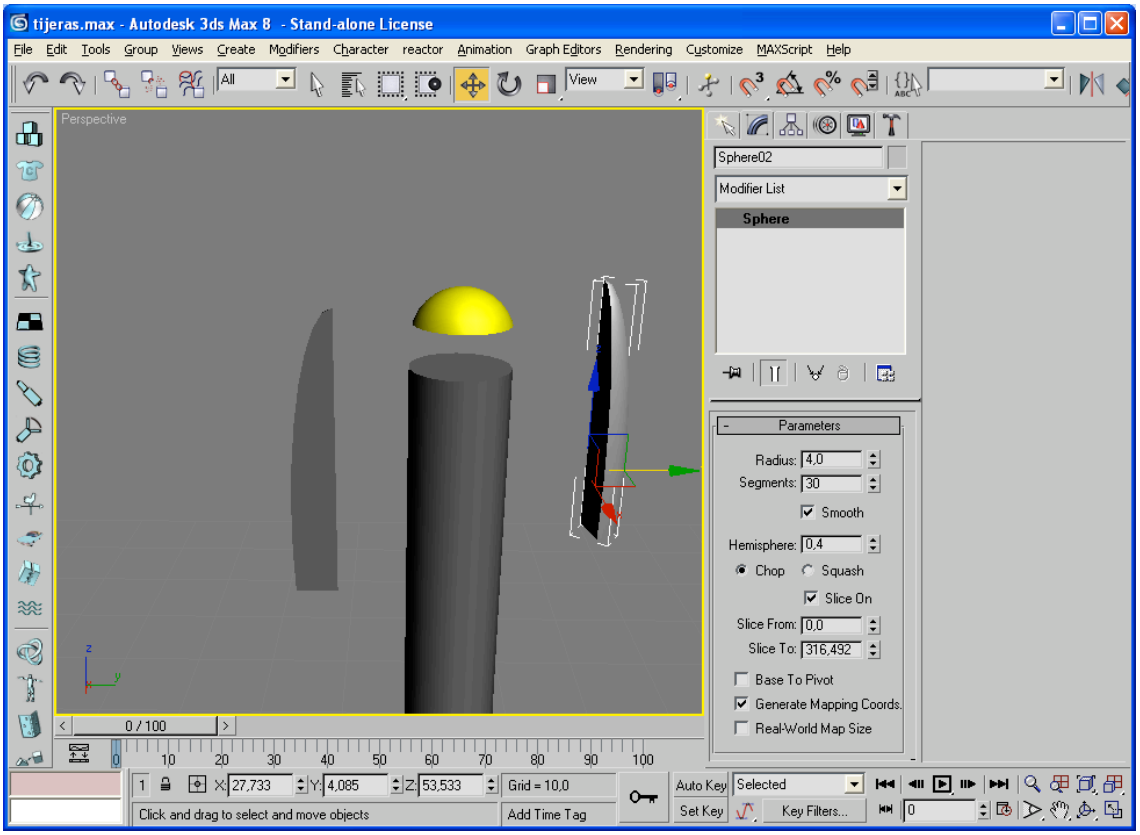
Como se puede apreciar en la figura, las tijeras están compuestas de 4 piezas que podríamos llamar Tubo, Esfera (para la articulación), Pinza1 y Pinza2.

Tubo se ha obtenido a partir del tipo de objeto Cylinder de 3DS Max. El resto de piezas: Esfera, Pinza1 y Pinza2 se han obtenido a partir de modificaciones del tipo de objeto Sphere.

En la siguiente imagen se observan los cuatro objetos 3D separados. Una de las pinzas aparece seleccionada y sus propiedades de modificación aparecen a la derecha dentro del marco Parameters.

Para una mejor lectura la siguiente tabla muestra los parámetros y sus valores:

Radius	4,0
Segments	30
Smooth	Activada
Hemisphere	0,4
Chop	Activada
Slice On	Activado
Slice From	0,0
Slice To	316,492
Base To Pivot	Desactivado
Generate Mapping Coords	Activado
Real-World Map Size	Desactivado



El diseño de cualquier herramienta, como se intuye a partir del ejemplo de las tijeras, puede realizarse fácilmente a partir de la creación de objetos simples y su posterior modificación. La optimización de estos diseños queda relegada a otro estudio, pero básicamente el objetivo es la minimización del número de polígonos.

Esto tan sólo pretende ser un ejemplo de metodología de diseño. Por supuesto, cualquier otro tipo de metodología para el diseño puede ser perfectamente válido.

Como se ha dejado entrever al hablar sobre la optimización, dado que las herramientas se integrarán en un sistema visual con detección de colisiones existe un compromiso en la relación detalle/coste computacional. En resumen, si queremos un gran nivel de detalle utilizaremos un elevado número de polígonos, por lo que el coste computacional será alto (fundamentalmente debido al cálculo de las colisiones). Por tanto, para conseguir un funcionamiento “suave” de la aplicación habrá que sacrificar el nivel de detalle y por consiguiente el número de polígonos de los objetos 3D empleados.

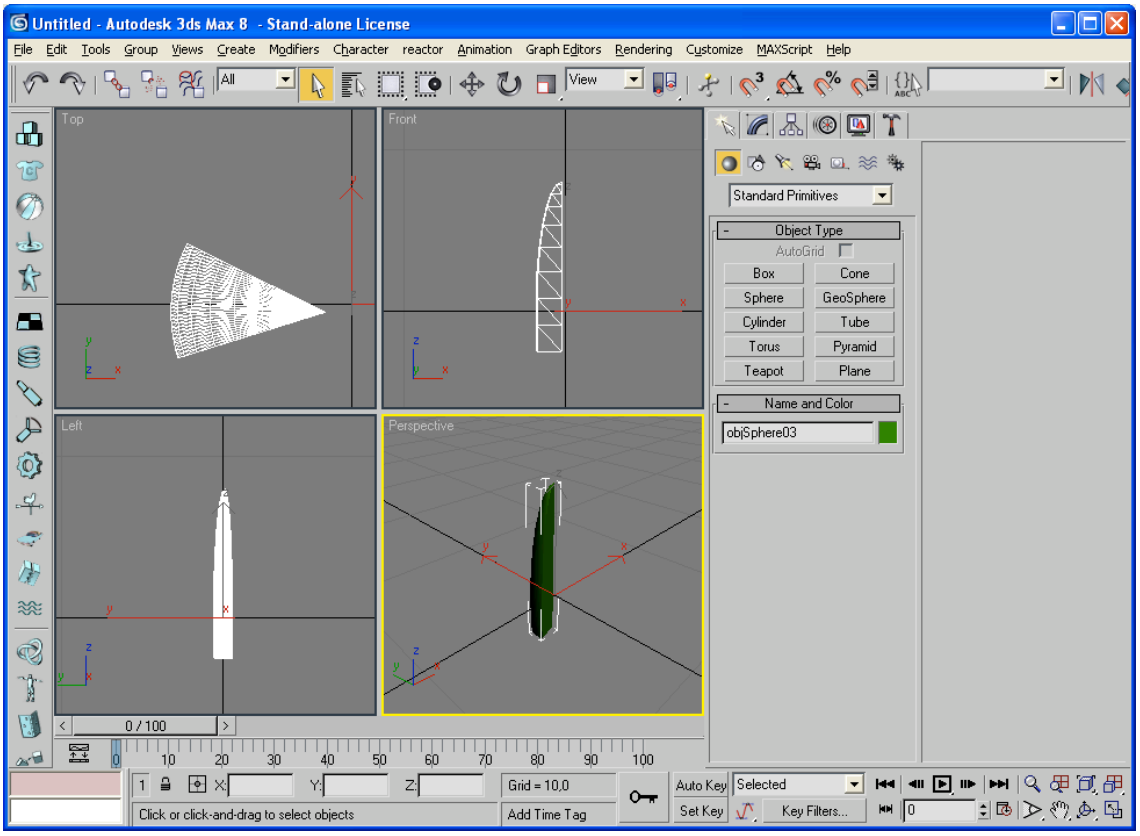
Almacenado de las piezas en un formato adecuado.

Al crear una herramienta con 3DS Max, el formato de trabajo por defecto es el de 3DS Max (archivos con la extensión .max). Este formato es muy adecuado para realizar el diseño de la herramienta y es una buena práctica guardar los trabajos en este formato.

Sin embargo, para poder utilizar las piezas diseñadas desde VTK, hay que trabajar con un formato diferente. Por supuesto, hay más de un formato diferente que se podría emplear en VTK pero en este proyecto se ha optado por el formato wavefront (archivos con la extensión .obj)

Para crear un objeto wavefront, se ha de utilizar las opciones “Export” y/o “Export selected” del menú y a continuación elegir en Tipo Wavefront Object (*.OBJ)

Para recuperar un objeto wavefront desde 3DS Max se ha de seleccionar la opción “Import” del menu File. En la siguiente imagen se puede ver Pinza1, despues de haber sido importada.



Además, antes de exportar el objeto 3D como objeto wavefront, para utilizar la pieza desde VTK deberemos posicionarla en el origen de coordenadas.

VTK soporta de forma muy sencilla la importación de archivos de tipo wavefront. En el siguiente apartado se explica como realizar dicha tarea.

Desarrollo de código para simular la herramienta.

Una vez se tienen creadas las piezas que componen la herramienta en formato wavefront, se ha de proceder al desarrollo de código. En este proyecto el lenguaje de programación es C++.

Como el modelado de una herramienta fija podría considerarse una simplificación del modelo de una herramienta articulada, se analizará primero la creación de una herramienta articulada para posteriormente, analizar las diferencias.

Herramientas Articuladas

Siguiendo con el ejemplo de las tijeras, y teniendo en cuenta que se ha creado la clase Herramienta3D, vamos a ver el código fuente de la clase Tijeras.

El primer punto a tener en cuenta es que la clase Tijeras hereda de la clase Herramienta3D.

La clase Herramienta3D proporciona métodos comunes para todas las herramientas para realizar operaciones como movimiento en azimuth y/o elevación, cargar el fichero OBJ para construir un actor VTK, etc.. El primer paso, por tanto, es declarar la clase de la herramienta como hija de Herramienta3D.

Para ello, en el archivo Tijeras.h:

```
class Tijeras: public Herramienta3D
```

En cuanto al archivo Tijeras.cpp, se compone de los siguientes métodos, que despues veremos en detalle:

- constructor de la clase
- destructor de la clase
- clicIzq
- clicDer
- AbrirCerrarPinzas
- RotarHerramienta

El constructor de la clase se encarga de realizar, mediante apoyo en la clase padre Herramienta3D, las siguientes operaciones:

- inicializaciones de objetos y variables necesarias
- asignación de nombres a cada una de las piezas y creación de actores (vtkActor) a partir de cada uno de los ficheros OBJ (generados en el apartado anterior)
- asociación con el sistema de detección de colisiones e inclusión de los objetos colisionables
- desactivación de la colisión entra las propias piezas de la herramienta
- posicionamiento de las piezas para formar la herramienta
- creación de dos ensamblados (vtkAssembly) para modelar la herramienta

Al tratarse de una herramienta articulada necesita del uso de dos ensamblados. Si por el contrario se desea modelar una herramienta fija sólo sería necesaria la creación de un ensamblado.

Veamos a continuación el código del constructor de la clase Tijeras:

```
Tijeras::Tijeras(CollisionObjectFactory* factory, int izq)
{
	std::cout << "Constructor de Tijeras\n";

	int posx;
	double* centro;
	char* objectName;
	vtkOBJReader* piezaOBJ;
	vtkPolyDataMapper **auxMapper;
	char** file;

	int componentsNumber = 4;
	char* objectsNameArray[] =
{"Cilindro","Esfera","Tijera1","Tijera2"};
```

```
	double scaleX=1;
	double scaleY=4;
	double scaleZ=1;
	double longTubo,longEsfera;

	// Declaraciones
	file = new char* [componentsNumber];
	file[0]="C:/vtkdata/Data/Viewpoint/cilindro.obj";
	file[1]="C:/vtkdata/Data/Viewpoint/esfera.obj";
	file[2]="C:/vtkdata/Data/Viewpoint/tijera1.obj";
	file[3]="C:/vtkdata/Data/Viewpoint/tijera2.obj";

	// Creamos los mappers
	auxMapper = new vtkPolyDataMapper* [componentsNumber];
	for (int i=0; i<componentsNumber; i++)	{
		auxMapper[i] = vtkPolyDataMapper::New();
	}

	// Asociamos el COF
	this->OCF=factory;

	// la posición puede ser derecha o izquierda
	if (izq==1) {
		posx=-100;
		}
	else {
		posx=100;
		}

	// Establecemos el número de actores y de ensamblados
	this->SetNumberOfActors(componentsNumber);
	this->SetNumberOfAssemblies(2);

	// establecemos las diferentes acciones asociadas a clic del
raton
	this->AddAction("rotar");
	this->AddAction("abrir");

	// Reservamos memoria para los actores e identificadores
	this->CreateDynamicArrays();

	// Establecemos la visibilidad/invisibilidad de los actores
	this->ActorVisibilityArray[0]=1;
	this->ActorVisibilityArray[1]=1;
	this->ActorVisibilityArray[2]=1;
	this->ActorVisibilityArray[3]=1;

	//leer archivo OBJ de actores y añadirlos al sistema de
colisiones
	for (int i=0; i<componentsNumber; i++)
	{
		objectName = objectsNameArray[i];
		piezaOBJ = this->ReadOBJFile(file[i]);
		this->OCF->AddObject(&this-
>ActorsVCIdArray[i],piezaOBJ,scaleX,scaleY,scaleZ,objectName,this-
>ActorsArray[i],&this->ActorsPAIdArray[i]);
		auxMapper[i]->SetInputConnection(this->OCF-
>GetFilterOfPiezasArray(this->ActorsPAIdArray[i])->GetOutputPort());
		this->ActorsArray[i]->SetMapper(auxMapper[i]);
		piezaOBJ->Delete();
```

```
            // Si es un actor invisible cambiamos el color
            if (ActorVisibilityArray[i]==0) ActorsArray[i]-
>GetProperty()->SetColor(0.0,1.0,1.0);
      }

      //medimos la longitud del tubo
      longTubo=ActorsArray[0]->GetYRange()[0]+ActorsArray[0]-
>GetYRange()[1];
      //medimos la longitud de la esfera
      longEsfera=ActorsArray[1]->GetYRange()[0]+ActorsArray[1]-
>GetYRange()[1];


      for (int i=0; i<componentsNumber; i++)
            {
                  centro=ActorsArray[i]->GetCenter();
                  ActorsArray[i]->SetOrigin(centro[0],0,centro[2]);
                  ActorsArray[i]->RotateWXYZ(-90,1,0,0);
            }

      // cambiamos el color de la esfera
      ActorsArray[1]->GetProperty()->SetColor(0.1,0.9,0.9);

      //desplazamos la esfera
      ActorsArray[1]->AddPosition(0,0,-longTubo*0.96);
      //desplazamos las pinzas la longitud del tubo
      ActorsArray[2]->AddPosition(0,0,-longTubo*0.96);
      ActorsArray[3]->AddPosition(0,0,-longTubo*0.96);

      // creamos un ensamblado con la esfera y las tapas
      // Crea un nuevo Assembly
      this->Ensamblado2 = vtkAssembly::New();
      // Añade los actores al Assembly
      for (int i=1; i<componentsNumber; i++)
      {
            this->Ensamblado2->AddPart(this->ActorsArray[i]);
      }

      //establece los angulos
      this->SetAnguloRotadoElevacion(0);
      this->SetAnguloRotadoAzimuth(0);

      // Desactiva colisiones
      this->OCF-
>DeactivateCollision(objectsNameArray[0],objectsNameArray[1]);
      this->OCF-
>DeactivateCollision(objectsNameArray[0],objectsNameArray[2]);
      this->OCF-
>DeactivateCollision(objectsNameArray[0],objectsNameArray[3]);
      this->OCF-
>DeactivateCollision(objectsNameArray[1],objectsNameArray[2]);
      this->OCF-
>DeactivateCollision(objectsNameArray[1],objectsNameArray[3]);
      this->OCF-
>DeactivateCollision(objectsNameArray[2],objectsNameArray[3]);

      // Crea un nuevo Assembly
      this->Ensamblado = vtkAssembly::New();

      // Añade los actores al Assembly
      this->Ensamblado->AddPart(this->ActorsArray[0]);
```

```
	this->Ensamblado->AddPart(this->Ensamblado2);

	this->Ensamblado->SetOrigin(0,30,200);
	centro=this->Ensamblado->GetCenter();
	//Ensamblado->RotateX(-90);
	Ensamblado->SetPosition(posx,100,centro[1]*2+500);
	// actualiza la matriz del ensamblado
	this->OCF->UpdateAssembliesMatrices(this->Ensamblado, this->Ensamblado2, this->ActorsVCIdArray, this->TransformArray);

	// delete

	for (int i=0; i<componentsNumber; i++)
	{
		auxMapper[i]->Delete();
		//delete file[i];
	}
	delete [] auxMapper;
	delete [] file;

}
```

El método RotarHerramienta se encarga de rotar el segundo ensamblado, que se corresponde con la parte articulada de la herramienta, actualizar las matrices en el sistema de detección de colisiones y comprobar si ha habido colisión por si se tiene que deshacer la operación de rotación:

```
void Tijeras::RotarHerramienta(float alfa)
{
	int colision=0;

	this->Ensamblado2->RotateZ(alfa);

	this->OCF->UpdateAssembliesMatrices(this->Ensamblado, this->Ensamblado2, this->ActorsVCIdArray, this->TransformArray);
	colision=this->AnyActorVisibleHasCollisioned();

	// si hay colision (de cualquier actor NO INVISIBLE) se revierte la operacion
	if (colision==1)
		{
		this->Ensamblado2->RotateZ(-alfa);
		this->OCF->UpdateAssembliesMatrices(this->Ensamblado, this->Ensamblado2, this->ActorsVCIdArray, this->TransformArray);
		}

}
```

El método AbrirCerrarPinzas se encarga de Abrir y Cerrar las tijeras. Para ello comprueba si no se excede de un ángulo máximo de apertura y/o cierre, y si no se excede se procede a rotar individualmente las piezas éxtremo Tijera1 y Tijera2, y si se

produce una colisión se deshace el movimiento, de igual forma que en el método RotarHerramienta.

```
void Tijeras::AbrirCerrarPinzas(float alfa)
{
	//por cada clic se abre un angulo alfa

	int colision=0;
	double angulo_actual=this->GetAnguloRotadoAbertura(); //obtiene
angulo
	double angulo=angulo_actual+alfa;  //calcula el nuevo angulo

	cout << "angulo="<<angulo<<"\n";
	//si el nuevo angulo es menor que el maximo permitido ejecuta el
movimiento
	if (angulo<40 && angulo>=7)
	  {

		this->ActorsArray[2]->RotateZ(alfa);
		this->ActorsArray[3]->RotateZ(-alfa);
		this->OCF->UpdateAssembliesMatrices(this->Ensamblado, this-
>Ensamblado2, this->ActorsVCIdArray, this->TransformArray);
		colision=this->AnyActorVisibleHasCollisioned();

		// si hay colision (de cualquier actor NO INVISIBLE) se
revierte la operacion
		if (colision==1)
			{
			this->ActorsArray[2]->RotateZ(-alfa);
			this->ActorsArray[3]->RotateZ(alfa);
			this->OCF->UpdateAssembliesMatrices(this->Ensamblado,
this->Ensamblado2, this->ActorsVCIdArray, this->TransformArray);
			}
		else
			{
			this->SetAnguloRotadoAbertura(angulo); //actualiza
angulo
			}
		} // hasta aqui si el angulo es permitido
}
```

En cuanto al método clicIzq, se encarga de ejecutar la operación de Abrir/Cerrar pinzas o de rotar la herramienta, en función de la acción correspondiente en el momento (para cambiar de acción se pulsa la 'a' del teclado).

```
void Tijeras::clicIzq()
{
	float alfa=0.5;
	if (strcmp(this->GetAction(),"abrir")==0) this-
>AbrirCerrarPinzas(alfa);
	else if (strcmp(this->GetAction(),"rotar")==0) this-
>RotarHerramienta(alfa);
}
```

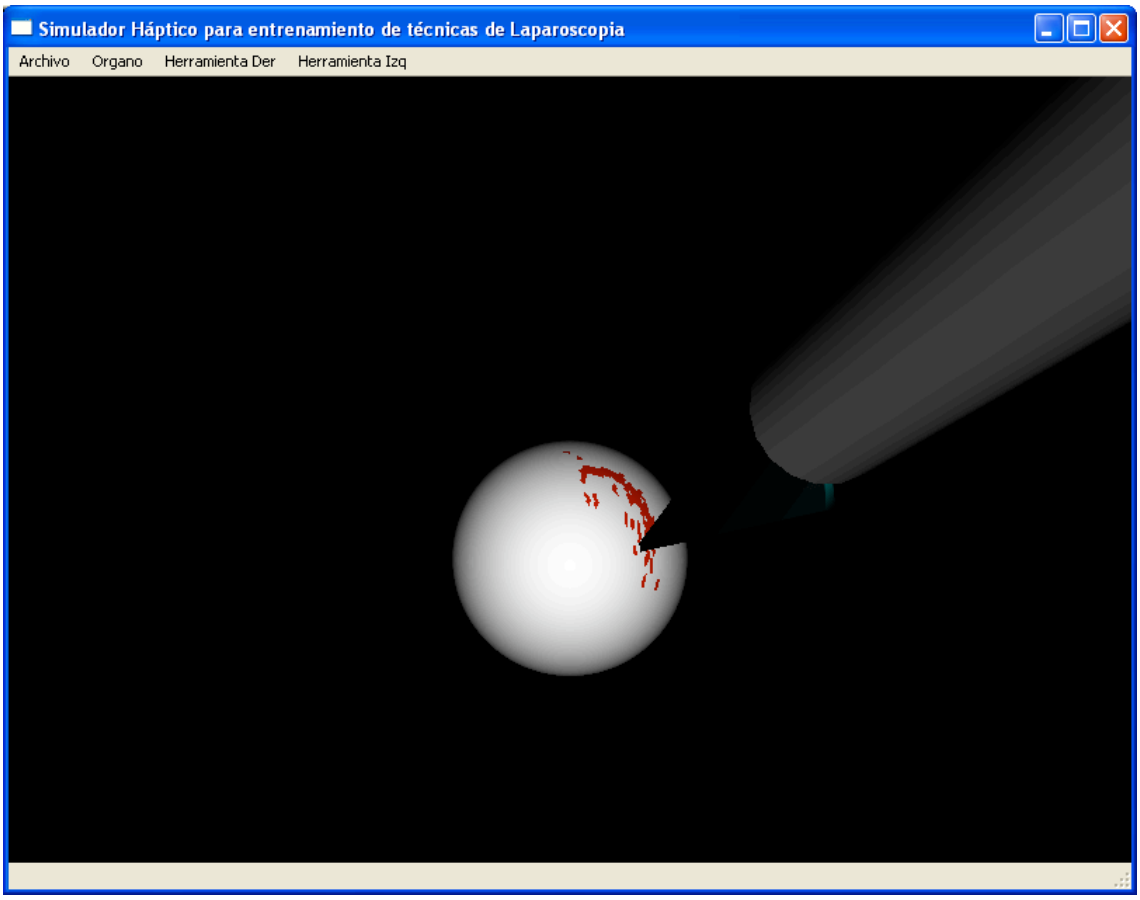
El método clicDer hace exactamente lo mismo pero en dirección opuesta. Es decir, en lugar de abrir, cerrar, y en lugar de rotar en un sentido, en el opuesto. A nivel de código la única diferencia es el factor alfa que pasa a ser -0.5.

Herramientas Fijas

Una vez visto el ejemplo de la herramienta articulada, la creación de una herramienta fija es similar salvo por las siguientes diferencias:

- se utiliza un único ensamblado
- para actualizar las rotaciones/movimientos de las piezas en el sistema de colisiones se utiliza el método `UpdateAssemblyMatrix` en lugar del método `UpdateAssembliesMatrices`

A continuación veremos el ejemplo de creación de un cutter tal y como se muestra en la figura:



Al igual que con la herramienta articulada, la herramienta fija tambien tiene que heredar de la clase Herramienta3D:

```
class Cutter: public Herramienta3D
```

En cuanto al archivo Cutter.cpp, se compone de los siguientes métodos, que a continuación veremos en detalle:

- constructor de la clase
- destructor de la clase
- clicIzq
- clicDer

El constructor de la clase se encarga de realizar, mediante apoyo en la clase padre Herramienta3D, las siguientes operaciones:

- inicializaciones de objetos y variables necesarias
- asignación de nombres a cada una de las piezas y creación de actores (vtkActor) a partir de cada uno de los ficheros OBJ (previamente generados)
- asociación con el sistema de detección de colisiones e inclusión de los objetos colisionables
- desactivación de la colisión entra las propias piezas de la herramienta
- posicionamiento de las piezas para formar la herramienta y establecimiento del origen de rotación
- creación de un ensamblado (vtkAssembly) para modelar la herramienta

Al tratarse de una herramienta fija tan sólo necesita del uso de un ensamblado. Si por el contrario se desea modelar una herramienta articulada, como vimos en el punto de Herramientas articuladas, necesitariamos la creación de dos ensamblados.

Veamos a continuación el código del constructor de la clase Cutter:

```
Cutter::Cutter(CollisionObjectFactory* factory, int izq)
{
	std::cout << "Constructor de Cutter\n";

	char* CUTTER="Cutter";
	char* FILO="Filo de Cutter";

	// Declaraciones
	char* file;
	char* objectName;
```

```
	vtkOBJReader* piezaOBJ;
	double scaleX,scaleY,scaleZ;
	vtkPolyDataMapper *auxMapper;
	int posx;

	this->OCF=factory;

	// la posición puede ser derecha o izquierda
	if (izq==1) {
		posx=-100;
		}
	else {
		posx=100;
		}

	this->SetNumberOfActors(2);

	// Reservamos memoria para los actores e identificadores
	this->CreateDynamicArrays();

	// Parámetros para escalado
	scaleX=1;
	scaleY=3;
	scaleZ=1;

	// Establecemos la visibilidad/invisibilidad de los actores
	this->ActorVisibilityArray[0]=1;
	this->ActorVisibilityArray[1]=0;

	//leer archivo OBJ de actores visibles y añadirlos al sistema de
colisiones
	file = "C:/vtkdata/Data/Viewpoint/cuchillo.obj";
	objectName = CUTTER;
	piezaOBJ = this->ReadOBJFile(file);
	this->OCF->AddObject(&this-
>ActorsVCIdArray[0],piezaOBJ,scaleX,scaleY,scaleZ,objectName,this-
>ActorsArray[0],&this->ActorsPAIdArray[0]);
	piezaOBJ->Delete();

	// Mapper y Actor para cuchillo
	auxMapper = vtkPolyDataMapper::New();
	auxMapper->SetInputConnection(this->OCF-
>GetFilterOfPiezasArray(this->ActorsPAIdArray[0])->GetOutputPort());
	this->ActorsArray[0]->SetMapper(auxMapper);

	//obtenemos la longitud del cuchillo
	double *lapizY=this->ActorsArray[0]->GetYRange();
	double longCutter = lapizY[1]-lapizY[0];

	centro=this->ActorsArray[0]->GetCenter();
	//quiero que el centro de las rotaciones sea la base del lapiz
	this->ActorsArray[0]->SetOrigin(centro[0],0,centro[2]);
	this->ActorsArray[0]->RotateWXYZ(-90,1,0,0);

	//establece los angulos
	this->SetAnguloRotadoElevacion(0);
	this->SetAnguloRotadoAzimuth(0);

	//actualiza la matriz en el sistema de colisiones
	this->OCF->UpdateActorMatrixInVCollide(this->ActorsVCIdArray[0],
this->ActorsArray[0]);
```

```
	cout << "crea filo de cutter..\n";

	//crea cuchilla invisible en la punta del cuchilla
	file = "C:/vtkdata/Data/Viewpoint/cuchillo_trans.obj";
	objectName = FILO;
	piezaOBJ = this->ReadOBJFile(file);
	this->OCF->AddObject(&this-
>ActorsVCIdArray[1],piezaOBJ,scaleX,scaleY,scaleZ,objectName,this-
>ActorsArray[1],&this->ActorsPAIdArray[1]);
	piezaOBJ->Delete();

	vtkPolyDataMapper *esferaM = vtkPolyDataMapper::New();
	esferaM->SetInputConnection(this->OCF-
>GetFilterOfPiezasArray(this->ActorsPAIdArray[1])->GetOutputPort());

	this->ActorsArray[1]->SetMapper(esferaM);

	this->ActorsArray[1]->SetOrigin(centro[0],0,centro[2]);
	this->ActorsArray[1]->RotateWXYZ(-90,1,0,0);

	this->ActorsArray[1]->GetProperty()->SetColor(0.0,1.0,1.0);

	//desactiva colision entre cutter y filo
	this->OCF->DeactivateCollision(CUTTER, FILO);

	//actualiza la matriz en el sistema de colisiones
	this->OCF->UpdateActorMatrixInVCollide(this->ActorsVCIdArray[1],
this->ActorsArray[1]);

	this->Ensamblado = vtkAssembly::New();
	this->Ensamblado->AddPart(this->ActorsArray[0]);
	centro=this->Ensamblado->GetCenter();
	this->Ensamblado->SetOrigin(centro[0],0,0);
	this->Ensamblado->AddPart(this->ActorsArray[1]);
	this->Ensamblado->SetPosition(posx,100,centro[1]*2+500);
}
```

En el caso del cutter, se compone de dos actores:

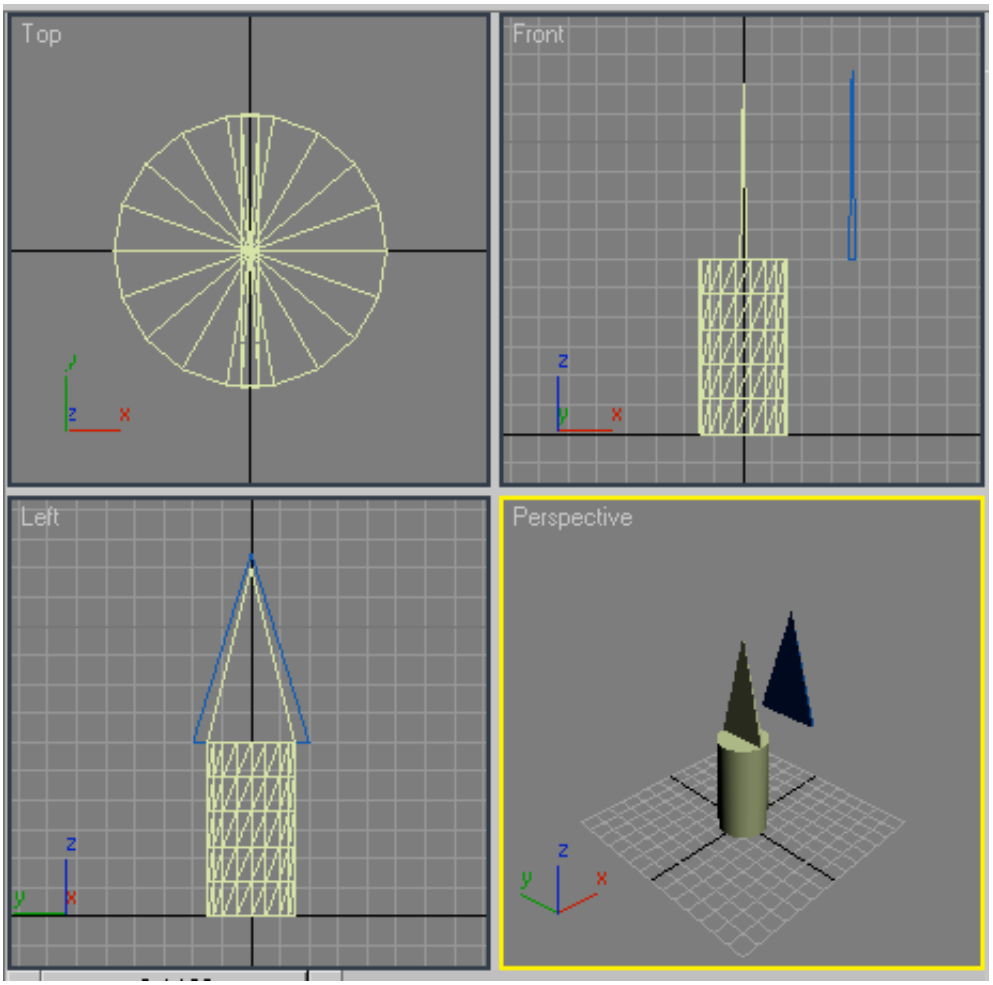
- Cutter
- Filo de cutter

Cutter es el objeto número cero de esta herramienta se utiliza como objeto visible y colisionable con el entorno. Son las colisiones de este objeto las que limitan el movimiento del cutter. Si este objeto colisiona realizando un movimiento en una determinada dirección, se deshace dicho movimiento, sin llegar a visualizarse por pantalla.

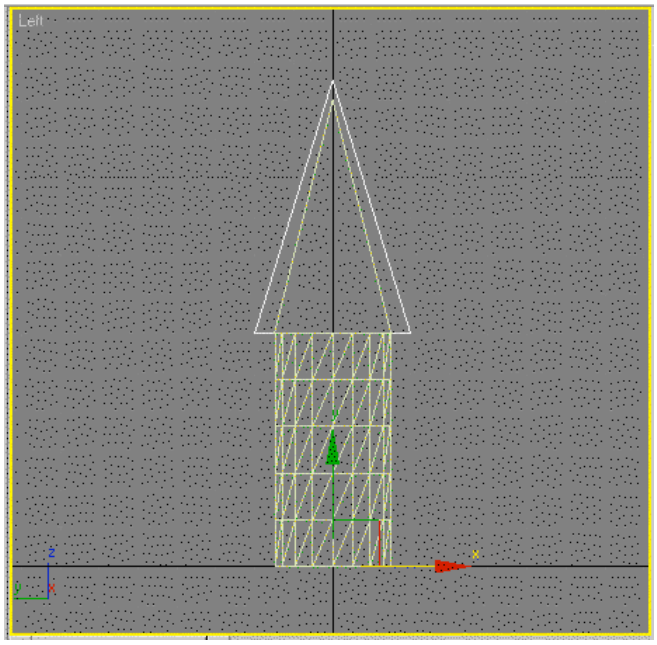
En cuanto a Filo de cutter, se establece su visibilidad a cero (`this->ActorVisibilityArray[1]=0`). Al establecer esta propiedad, sus colisiones no

afectan al movimiento de la herramienta, en este sentido, se trata como si fuera un objeto invisible. Sin embargo, su colisión importa, ya que para que el cutter haga sangrar un órgano, lo que se está haciendo realmente es pintar de rojo los polígonos colisionados del órgano con este objeto invisible. Al haberse diseñado como una funda del filo del cutter se consigue el efecto de una ligera penetración en el objeto como si cortara.

A continuación, en la siguiente imagen, se aprecian ambos objetos. El objeto Filo de Cutter se ha separado una distancia a lo largo del eje X para poder diferenciarlos.



En la siguiente imagen, se observa en detalle la vista Left, donde se puede comprobar que Filo de Cutter sobresale por encima de las dimensiones de Cutter. Este es el motivo por el que se consigue el efecto de ligera penetración en el órgano.



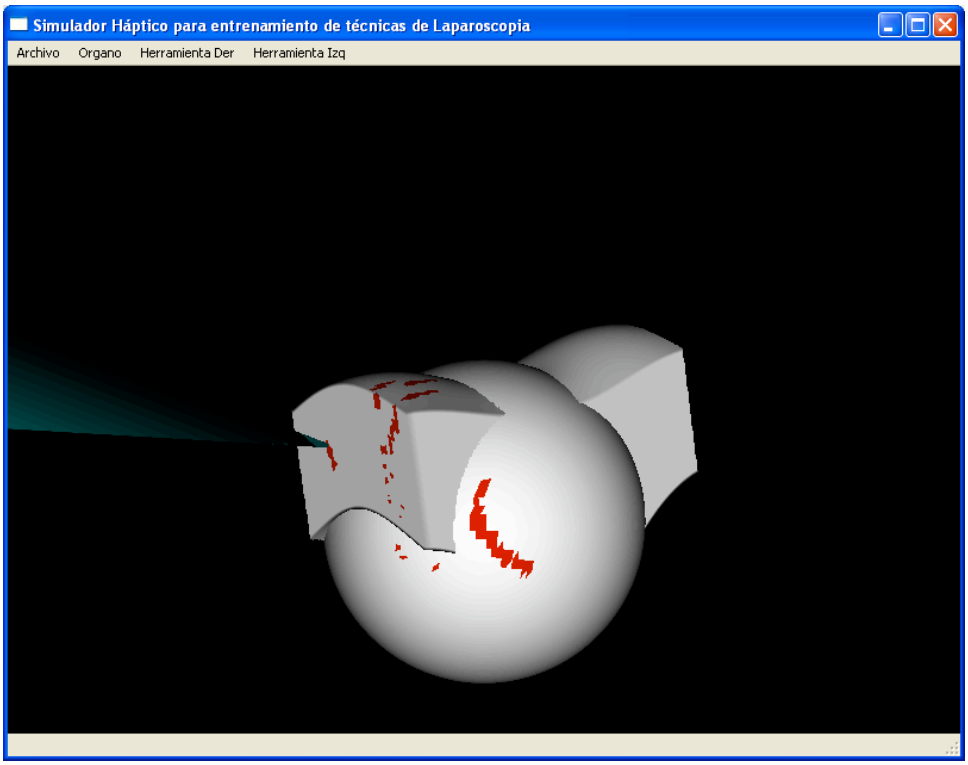
El objeto Filo de cutter es lo que en comentarios del código anterior aparece como objeto invisible. La finalidad de este objeto invisible en el caso del Cutter, como se comentó previamente, es la siguiente: al colisionar con un órgano, si se hace clic con el botón izquierdo del ratón, los polígonos intersectados del órgano se pintan de rojo simulando así la sangre.

A continuación se muestra el código fuente del método clicIzq, el encargado de pintar. Básicamente, lo que hace este método es comprobar si ha habido colisión entre el objeto invisible y algún órgano.

```
void Cutter::clicIzq()
{
      this->OCF->InformAboutCollisions();
      // obtiene el id del organo colisionado
      int organo_PA_Id=this->OCF-
>GetOrganoCollisionedWithHerramienta(this->ActorsVCIdArray[1]);
      // comprueba si ha colisionado el objeto invisible y si ademas
ha colisionado con un organo
      if (this->OCF->HasCollisioned(this-
>ActorsVCIdArray[1])&&organo_PA_Id!=-1)
        {
            //tiene que pintar
            int numCeldas;
            scalars=vtkUnsignedCharArray::SafeDownCast(this->OCF-
>GetFilterOfPiezasArray(organo_PA_Id)->GetOutput()->GetCellData()-
>GetScalars());
            if (scalars==NULL)
                  {
```

```
                  numCeldas=this->OCF-
>GetFilterOfPiezasArray(organo_PA_Id)->GetOutput()-
>GetNumberOfCells();
                  std::cout << "generando scalars..para numceldas=" <<
numCeldas << std::endl;
                  scalars=vtkUnsignedCharArray::New();
                  scalars->SetNumberOfComponents(4);
                  scalars->SetNumberOfTuples(numCeldas);
                  float r=0;float g=0;float b=0;
                  for (int i=0;i<numCeldas;i++)
                        {
                        scalars->SetTuple4(i,255,255,255,255);
                        }
                  this->OCF->GetFilterOfPiezasArray(organo_PA_Id)-
>GetOutput()->GetCellData()->SetScalars(scalars);

                  }
            this->OCF->GetFilterOfPiezasArray(organo_PA_Id)-
>GetOutput()->GetCellData()->Modified();
            //PINTA
            int obj_pair_num=this->OCF-
>GetCollisionedPairOfObjectsNum(this->ActorsVCIdArray[1]);
            int num_tri;
            // para saber si se ha de colorear tri1ID o tri2ID
            int obj_id=this->OCF->report.obj1ID(obj_pair_num);
            bool tri1=false;
            if (obj_id==organo_PA_Id) tri1=true;
            else tri1=false;

            for (int k=0;k<this->OCF-
>report.numTriPairs(obj_pair_num);k++)
                  {
                        if (tri1==true) {
                              num_tri=this->OCF-
>report.tri1ID(obj_pair_num,k);
                        }
                        else {
                              num_tri=this->OCF-
>report.tri2ID(obj_pair_num,k);
                        }

                        // pinta de color rojo
                        scalars->SetTuple4(num_tri,220,17,17,255);
                        }
            this->OCF->GetFilterOfPiezasArray(organo_PA_Id)-
>GetOutput()->GetCellData()->Modified();
            }
      else std::cout << "No puede pintar.\n";

}
```

A continuación se muestra una imagen en la que el cutter ha pintado (“cortado”) varias órganos.



La clase Herramienta3D

Como hemos podido comprobar en los anteriores apartados donde se hablaba sobre herramientas articuladas y herramientas fijas, ambos tipos de herramientas tienen en comun, a nivel de código fuente, que heredan de la misma clase Herramienta3D.

La clase Herramienta3D se compone de los siguientes métodos:

- constructor de la clase
- destructor de la clase
- AddAction
- GetAction
- Azimuth
- Elevation
- ruedaDelante
- ruedaAtras
- AddPositionXYZ
- CreateDynamicArrays

- DeleteDynamicArrays
- RotateXYZAssembly
- AnyActorVisibleHasCollisioned
- ReadOBJFile
- teclaPulsada

A continuación se ofrece una explicación de la funcionalidad de dichos métodos.

El constructor se encarga de realizar las inicializaciones necesarias y el destructor se encarga de liberar memoria de los objetos inicializados.

Una acción es una operación caracteristica que realiza una herramienta, por ejemplo, rotar, pintar, grapar son ejemplos de acciones. Dado que cada herramienta puede tener cero o más acciones, para añadir una acción (realmente se añade el texto que indica qué accion es capaz de realizar) se utiliza el método AddAction. En cuanto al método GetAction devuelve el texto de la acción asociada al clic de ratón en un determinado momento.

Para herramientas que tengan más de una acción asociada, la forma de cambiar de acción es pulsando la tecla ‘a’. El cambio de acción se realiza de forma cíclica. Por ejemplo, para una herramienta que tenga asociadas tres acciones en el siguiente orden: abrir, rotar y grapar. Al hacer clic con el ratón la acción que ejecutará la herramienta será la de abrir. Si se pulsa ‘a’ entonces la acción será rotar. Si se vuelve a pulsar ‘a’ la acción será grapar. Al haber llegado a la última acción, al pulsar ‘a’ la acción seleccionada será la primera, es decir, abrir, y así sucesivamente. Si se quiere saber qué acción tiene asociada una herramienta en un determinado instante se puede pulsar ‘i’.

Azimuth y Elevation son los métodos que se encargan de mover la herramienta en azimuth y elevación respectivamente. Antes de mostrar el movimiento en pantalla, se comprueba si al mover la herramienta se produce alguna colisión entre cualquier objeto visible de la herramienta. En el caso de que haya colisión de algún objeto visible, se deshace el movimiento y se vuelve a la posición que tenía la herramienta antes de efectuar el movimiento.

RuedaDelante y ruedaAtras se encargan de calcular los desplazamientos que tiene que efectuar la herramienta hacia delante y hacia atrás respectivamente apoyándose en el método AddPositionXYZ.

El resto de métodos son utilizados por los propios métodos de la clase o clases hijas y son utilizados para diferentes tareas como inicialización/destrucción de objetos, para saber si ha colisionado algún objeto visible, para controlar la acción asociada a la pulsación de una tecla o para la lectura de un fichero wavefront.

A continuación se muestra el código fuente de la clase Herramienta3D:

```
#include "Herramienta3D.h"

Herramienta3D::Herramienta3D(void)
{
	std::cout << "Constructor Herramienta3D\n";
	this->tr = vtkTransform::New();
	this->SetAnguloRotadoAzimuth(0.0);
	this->SetAnguloRotadoElevacion(0.0);
	this->SetAnguloRotadoAbertura(7.0);
	this->SetNumberOfAssemblies(1);
	this->OCF=NULL;
	this->SetNumberOfActors(0);
	this->actionsArray[0]=NULL;
	this->actionsArray[1]=NULL;
	this->actionsArray[2]=NULL;
	// inicializamos el iterador de acciones (clic izqdo)
	this->actionsArrayIterator=0;
	// comprueba max y minimos
	this->comprobarAngulo=true;

}

Herramienta3D::~Herramienta3D(void)
{
	std::cout << "Destructor Herramienta3D\n";
	this->DeleteDynamicArrays();
	tr->Delete();
}

void Herramienta3D::AddAction(char* accion)
{
	for (int i=0;i<3;i++)
		{
		if (this->actionsArray[i]==NULL)
			{
			this->actionsArray[i]=accion;
			cout << "La accion "<<i<<" de esta Herramienta es:
"<<actionsArray[i]<<"\n";
			break;
			}
		}
```

```
}

char* Herramienta3D::GetAction()
{
	if (this->actionsArray[this->actionsArrayIterator]==NULL) return
"none";
	return this->actionsArray[this->actionsArrayIterator];
}

void Herramienta3D::Azimuth(float z)
{
	int colision=0;

	// para corregir saltos grandes debido a que el raton se mueva
por fuera de la ventana o por que no esté inicializado
	if (abs(z)>8) z=0;

	//obtiene angulo
	double angulo_actual=this->GetAnguloRotadoAzimuth();

	//calcula el nuevo angulo
	double angulo=angulo_actual-z;

	//si el nuevo angulo es menor que el maximo permitido ejecuta el
movimiento
	if ((angulo<90&&angulo>-90) || true)
	  {
		// mueve el ensamblado y actualiza las matrices de sus
actores en vcollide
		this->RotateXYZAssembly(0,-z,0);
		colision=this->AnyActorVisibleHasCollisioned();

		// si hay colision (de cualquier actor VISIBLE) se revierte
la operacion
		if (colision==1)
			{
			// mueve el ensamblado y actualiza las matrices de
sus actores en vcollide
			this->RotateXYZAssembly(0,z,0);
			}
		else
			{
			this->SetAnguloRotadoAzimuth(angulo); //actualiza
angulo
			}

		} // hasta aqui si el angulo es permitido
}

void Herramienta3D::Elevation(float z)
{

	int colision=0;
	if (abs(z)>8) z=0; // para corregir saltos grandes debido a que
el raton se mueva por fuera de la ventana o por que no esté
inicializado
	double angulo_actual=this->GetAnguloRotadoElevacion(); //obtiene
angulo
	double angulo=angulo_actual-z;	//calcula el nuevo angulo
```

```
      //si el nuevo angulo es menor que el maximo permitido ejecuta el
movimiento
      if (angulo<90&&angulo>-90)
        {
            this->RotateXYZAssembly(z,0,0); //Mueve los actores y
actualiza sus matrices en VCollide
            colision=this->AnyActorVisibleHasCollisioned();

            // si hay colision (de cualquier actor VISIBLE) se revierte
la operacion
            if (colision==1)
                  {
                  this->RotateXYZAssembly(-z,0,0);
                  }
            else
                  {
                  this->SetAnguloRotadoElevacion(angulo); //actualiza
angulo
                  //std::cout << "Elevacion: "<<this-
>GetAnguloRotadoElevacion()<<std::endl;
                  }
            } // hasta aqui si el angulo es permitido
}

void Herramienta3D::ruedaDelante()
{
      int r=2;
      float phi=180+this->GetAnguloRotadoAzimuth();
      phi=phi*vtkMath::DegreesToRadians();
      float theta=90+this->GetAnguloRotadoElevacion();
      theta=theta*vtkMath::DegreesToRadians();
      float x=r*sin(theta)*sin(phi);
      float y=r*cos(theta);
      float z=r*sin(theta)*cos(phi);
      this->AddPositionXYZ(x,y,z);
      //std::cout << "XYZ("<<x<<","<<y<<","<<z<<")"<<std::endl;
}
void Herramienta3D::ruedaAtras()
{
      int r=-2;
      float phi=180+this->GetAnguloRotadoAzimuth();
      phi=phi*vtkMath::DegreesToRadians();
      float theta=90+this->GetAnguloRotadoElevacion();
      theta=theta*vtkMath::DegreesToRadians();
      float x=r*sin(theta)*sin(phi);
      float y=r*cos(theta);
      float z=r*sin(theta)*cos(phi);
      this->AddPositionXYZ(x,y,z);
      //std::cout << "XYZ("<<x<<","<<y<<","<<z<<")"<<std::endl;
}

void Herramienta3D::AddPositionXYZ(float x, float y, float z)
{
      // obtenemos el numero de ensamblados que componen la
herramienta
      int numAssemblies = this->GetNumberOfAssemblies();

      // hace avanzar o retroceder el ensamblado
      this->Ensamblado->AddPosition(x,y,z);

      //actualiza las matrices del ensamblado en VCollide
```

```
	if (numAssemblies==1) this->OCF->UpdateAssemblyMatrix(this->GetNumberOfAssemblies(),this->Ensamblado, this->ActorsVCIdArray, this->TransformArray);
	else this->OCF->UpdateAssembliesMatrices(this->Ensamblado,this->Ensamblado2, this->ActorsVCIdArray, this->TransformArray);

	//si ha chocado revierte la operacion
	int colision=this->AnyActorVisibleHasCollisioned();

	// si hay colision (de cualquier actor NO INVISIBLE) se revierte la operacion
	if (colision==1)
		{
		cout << "Deshaciendo movimiento...\n";
		//this->clicIzq();
		this->Ensamblado->AddPosition(-x,-y,-z);
		//actualiza las matrices del ensamblado en VCollide
		if (numAssemblies==1) this->OCF->UpdateAssemblyMatrix(this->GetNumberOfAssemblies(),this->Ensamblado, this->ActorsVCIdArray, this->TransformArray);
		else this->OCF->UpdateAssembliesMatrices(this->Ensamblado,this->Ensamblado2, this->ActorsVCIdArray, this->TransformArray);
		}
}


void Herramienta3D::CreateDynamicArrays()
{
	int number = this->GetNumberOfActors();

	if (number!=0)
		{
		this->ActorsArray = new vtkActor* [number];
		this->TransformArray = new vtkTransform* [number];

		for (int j=0; j<number; j++) {
			this->ActorsArray[j] = vtkActor::New();
			this->TransformArray[j] = vtkTransform::New();
			}
		this->ActorsVCIdArray = new int[number];
		this->ActorsPAIdArray = new int[number];
		this->ActorVisibilityArray = new int[number];
		}
}

void Herramienta3D::DeleteDynamicArrays()
{
	int number = this->GetNumberOfActors();

	if (number!=0)
		{
		for (int j=0; j<number; j++) {
			ActorsArray[j]->Delete();
			TransformArray[j]->Delete();
			}
		delete [] this->ActorsArray;
		delete [] this->TransformArray;
		delete [] this->ActorsVCIdArray;
		delete [] this->ActorsPAIdArray;
		delete [] this->ActorVisibilityArray;
```

```
            }
}



void Herramienta3D::RotateXYZAssembly(float x, float y, float z)
{
      //Rota el ensamblado principal
      this->Ensamblado->RotateY(y);
      this->Ensamblado->RotateX(x);
      // Actualiza las matrices de los actores que componen el/los
ensamblado/s
      if (this->GetNumberOfAssemblies()==1)
            {
            this->OCF->UpdateAssemblyMatrix(this-
>GetNumberOfAssemblies(),this->Ensamblado, this->ActorsVCIdArray,
this->TransformArray);
            }
      else
            {
            this->OCF->UpdateAssembliesMatrices(this->Ensamblado,this-
>Ensamblado2, this->ActorsVCIdArray, this->TransformArray);
            }

}


int Herramienta3D::AnyActorVisibleHasCollisioned()
{
      // Calcula colisiones
      this->OCF->CalculateCollisions();

      int idActor;
      int* punteroActorId=this->ActorsVCIdArray;
      vtkActor* actor;

      for (int i=0;i<this->GetNumberOfActors();i++)
            {
            idActor=*(punteroActorId+i);
            actor=this->ActorsArray[i];
            // Devuelve 1 si algun actor "visible" de la herramienta ha
colisionado
            if (this->OCF->HasCollisioned(idActor) && this-
>ActorVisibilityArray[i]==1)
                  return 1;
            }

      return 0;

}


vtkOBJReader* Herramienta3D::ReadOBJFile(char *file)
{
      this->wavefront = vtkOBJReader::New();
      this->wavefront->SetFileName(file);
      this->wavefront->Update();
      return this->wavefront;
}

void Herramienta3D::teclaPulsada(char tecla)
```

```
{

	if (tecla=='a' || tecla=='A')
	{
		// cambia de accion
		this->actionsArrayIterator++;

		// establecido el máximo de acciones a 3
		actionsArrayIterator=actionsArrayIterator%3;

		if (this->actionsArray[actionsArrayIterator]==NULL)
			{
			actionsArrayIterator=0;
			}
		cout << "Accion seleccionada="<<this->GetAction()<<"\n";
	}
	else if (tecla=='i' || tecla=='I') {
		cout << "Informacion: actualmente la accion es "<<this-
>GetAction()<<"\n";
	}

}
```

Capítulo 6. Simulador LAPUMH

Interfaz Visual: QT con VTK

Qt es una biblioteca multiplataforma para desarrollar interfaces gráficas de usuario. Fue creada por la compañía noruega Trolltech. Qt es utilizada en KDE, un entorno de escritorio para sistemas como GNU/Linux o FreeBSD, entre otros. Utiliza el lenguaje de programación C++ de forma nativa y además existen bindings para C, Python (PyQt), Java (Qt Jambi), Perl (PerlQt),Gambas (gb.qt), Ruby (QtRuby), PHP (PHP-Qt) y Mono (Qyoto) entre otros.

Para enlazar QT y VTK se ha utilizado QVTK. La ventana de renderizado VTK se ha conectado con QT a traves de la clase QVTKWidget.

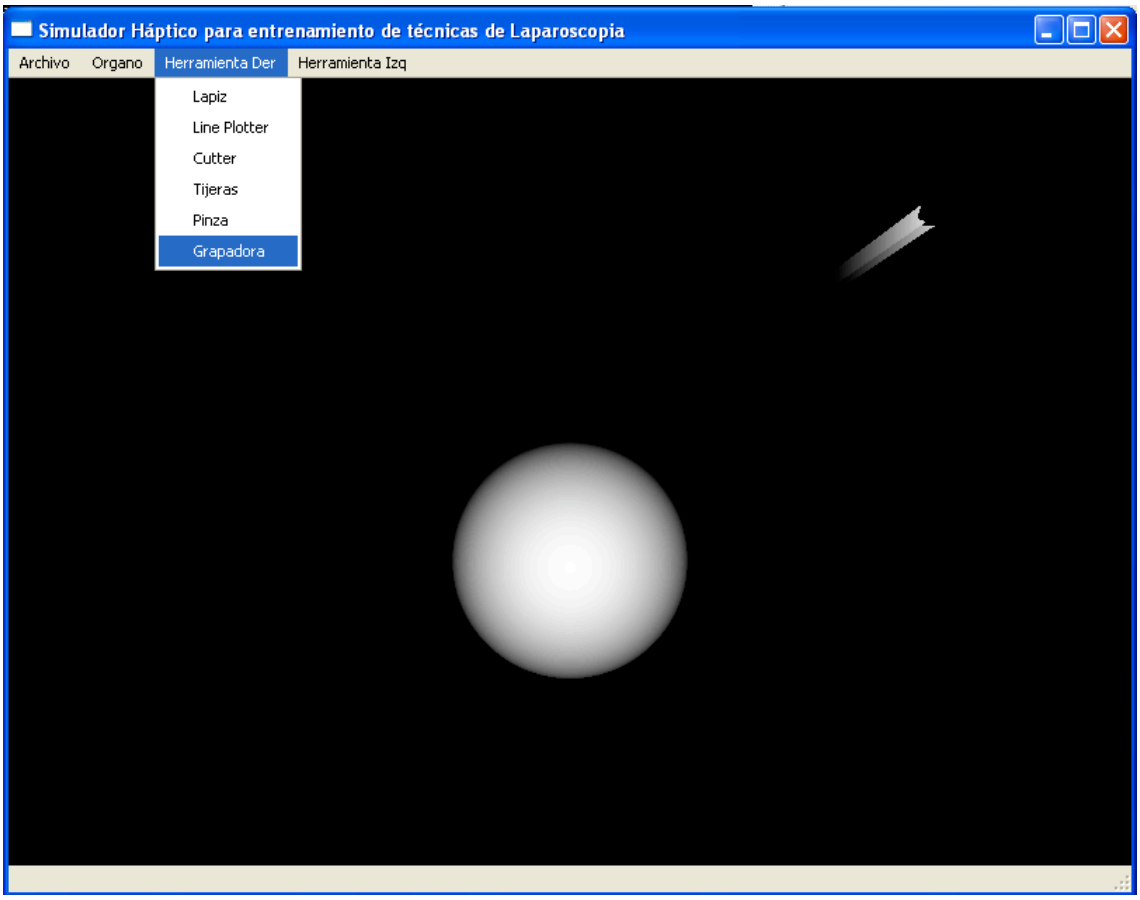
A continuación se muestra parte del código fuente de la clase Principal, en concreto, la parte encargada de la inicialización de la instancia QVTKWidget y su asociación con la ventana de renderizado VTK:

```
stage = new QVTKWidget();
setCentralWidget(stage);
stage->GetRenderWindow()->AddRenderer(ren);
stage->GetRenderWindow()->SetInteractor(iren);
stage->GetRenderWindow()->Render();
```

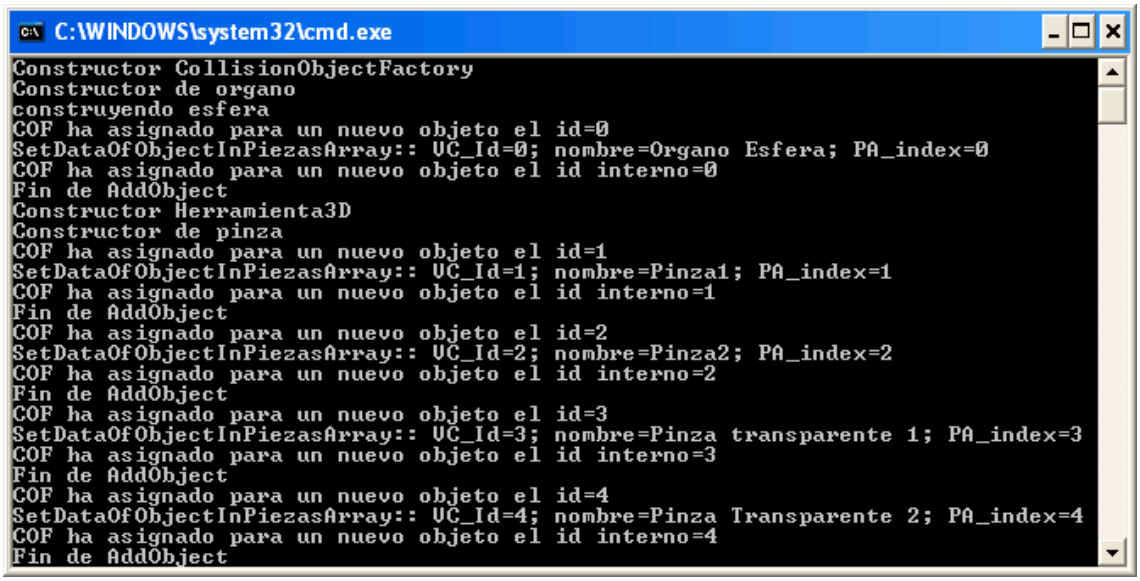
Aspecto Interfaz

El aspecto de la interfaz es sencillo, consta de una barra de menú con los siguientes menu: Archivo, Organo, Herramienta Der y Herramienta Izq.

A continuación se muestra una captura de pantalla:



Al ejecutar la aplicación se abre otra ventana donde aparece una consola de texto y en la que aparecen mensajes de carácter informativo:

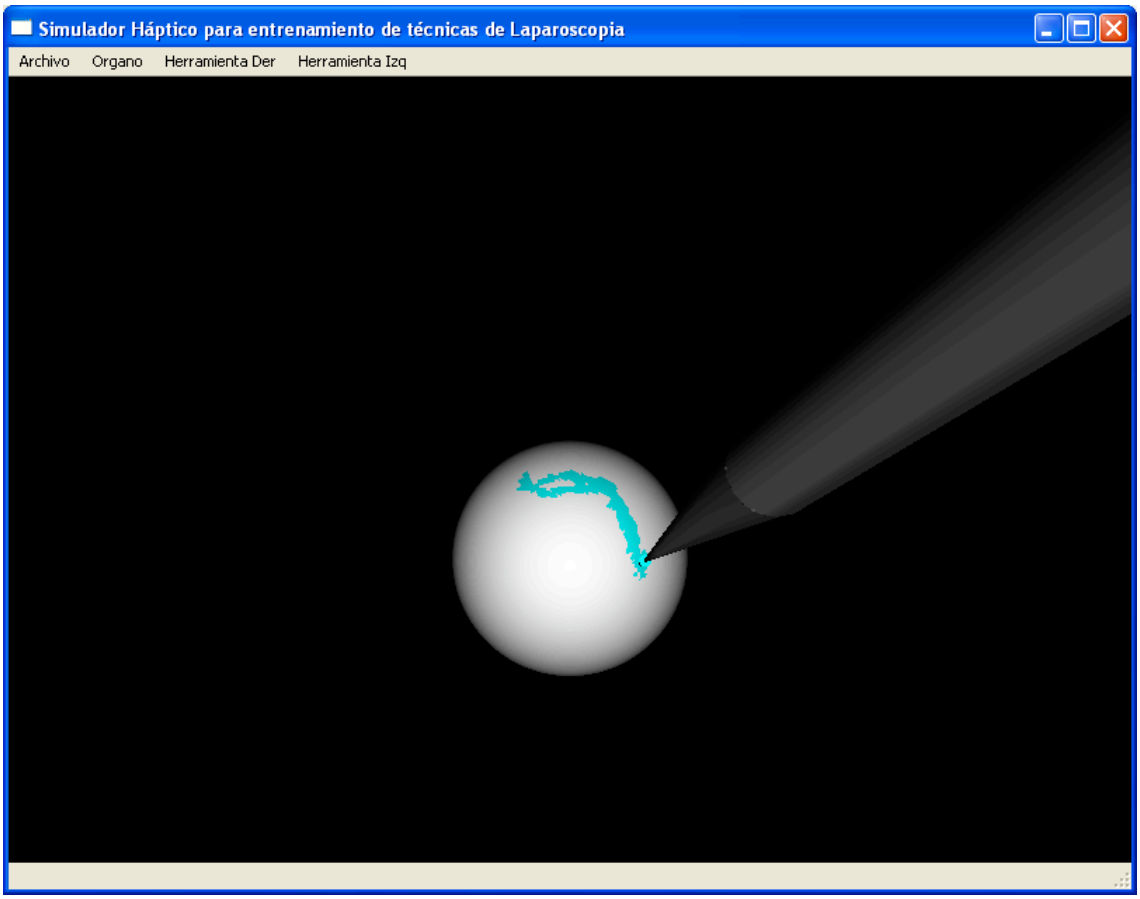

C:\WINDOWS\system32\cmd.exe
Constructor CollisionObjectFactory
Constructor de organo
construyendo esfera
COF ha asignado para un nuevo objeto el id=0
SetDataOfObjectInPiezasArray:: VC_Id=0; nombre=Organo Esfera; PA_index=0
COF ha asignado para un nuevo objeto el id interno=0
Fin de AddObject
Constructor Herramienta3D
Constructor de pinza
COF ha asignado para un nuevo objeto el id=1
SetDataOfObjectInPiezasArray:: VC_Id=1; nombre=Pinza1; PA_index=1
COF ha asignado para un nuevo objeto el id interno=1
Fin de AddObject
COF ha asignado para un nuevo objeto el id=2
SetDataOfObjectInPiezasArray:: VC_Id=2; nombre=Pinza2; PA_index=2
COF ha asignado para un nuevo objeto el id interno=2
Fin de AddObject
COF ha asignado para un nuevo objeto el id=3
SetDataOfObjectInPiezasArray:: VC_Id=3; nombre=Pinza transparente 1; PA_index=3
COF ha asignado para un nuevo objeto el id interno=3
Fin de AddObject
COF ha asignado para un nuevo objeto el id=4
SetDataOfObjectInPiezasArray:: VC_Id=4; nombre=Pinza Transparente 2; PA_index=4
COF ha asignado para un nuevo objeto el id interno=4
Fin de AddObject

Herramientas software

Las herramientas que se han modelado en la aplicación son las siguientes:

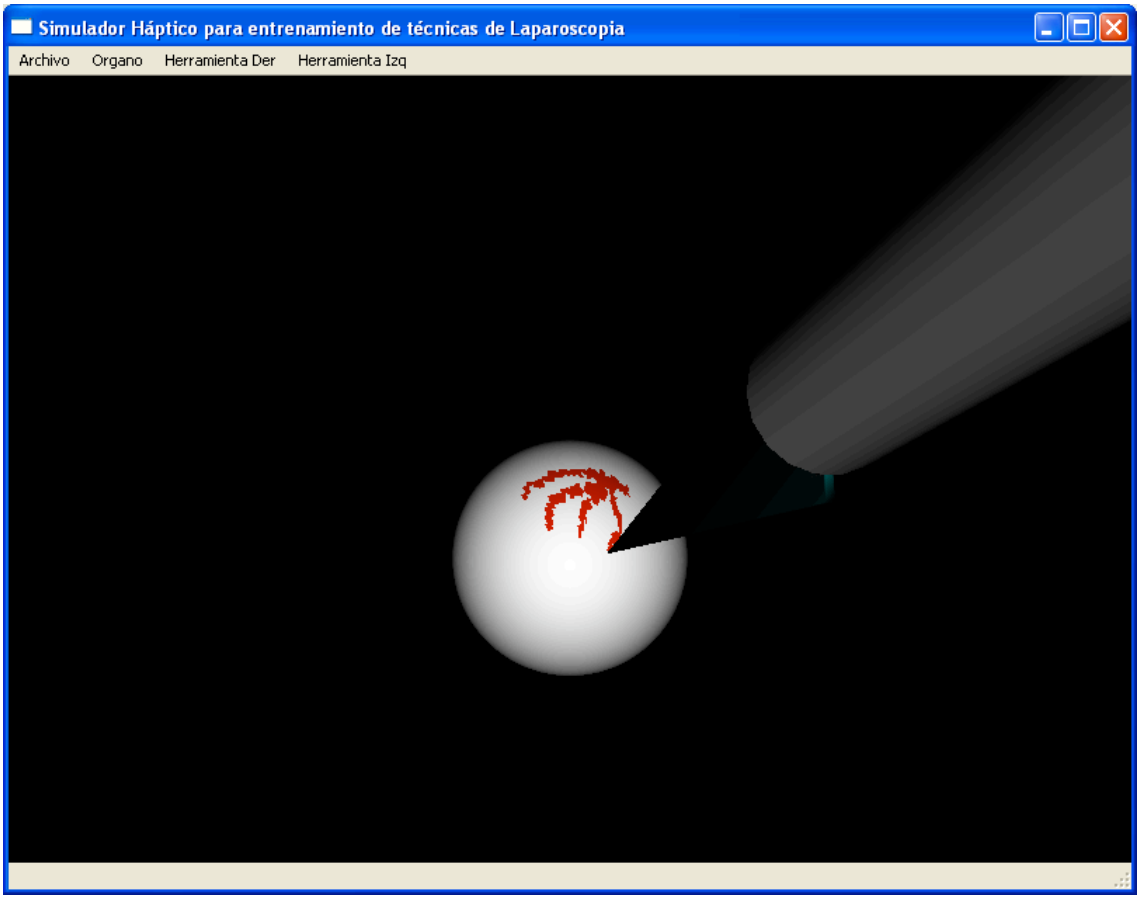
- Lapiz
- Cutter
- Tijeras
- Pinzas
- Grapadora

El lapiz, se trata de una herramienta fija, que cuando entra en contacto (su punta) con un órgano lo pinta de color azul. En la siguiente imagen se observa dicho comportamiento:

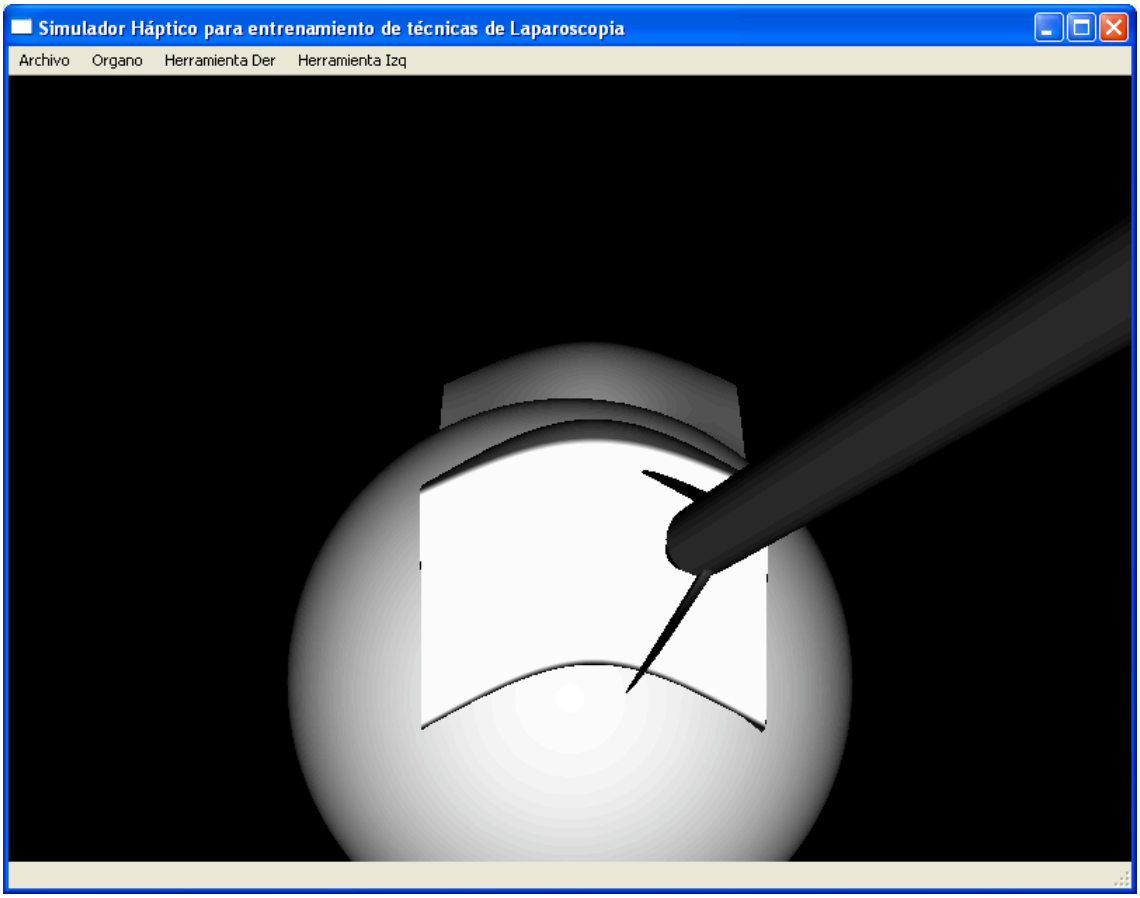


Para mover el lapiz por el entorno se utiliza el movimiento del ratón. Para avanzar o retroceder en profundidad se utiliza la rueda del ratón.

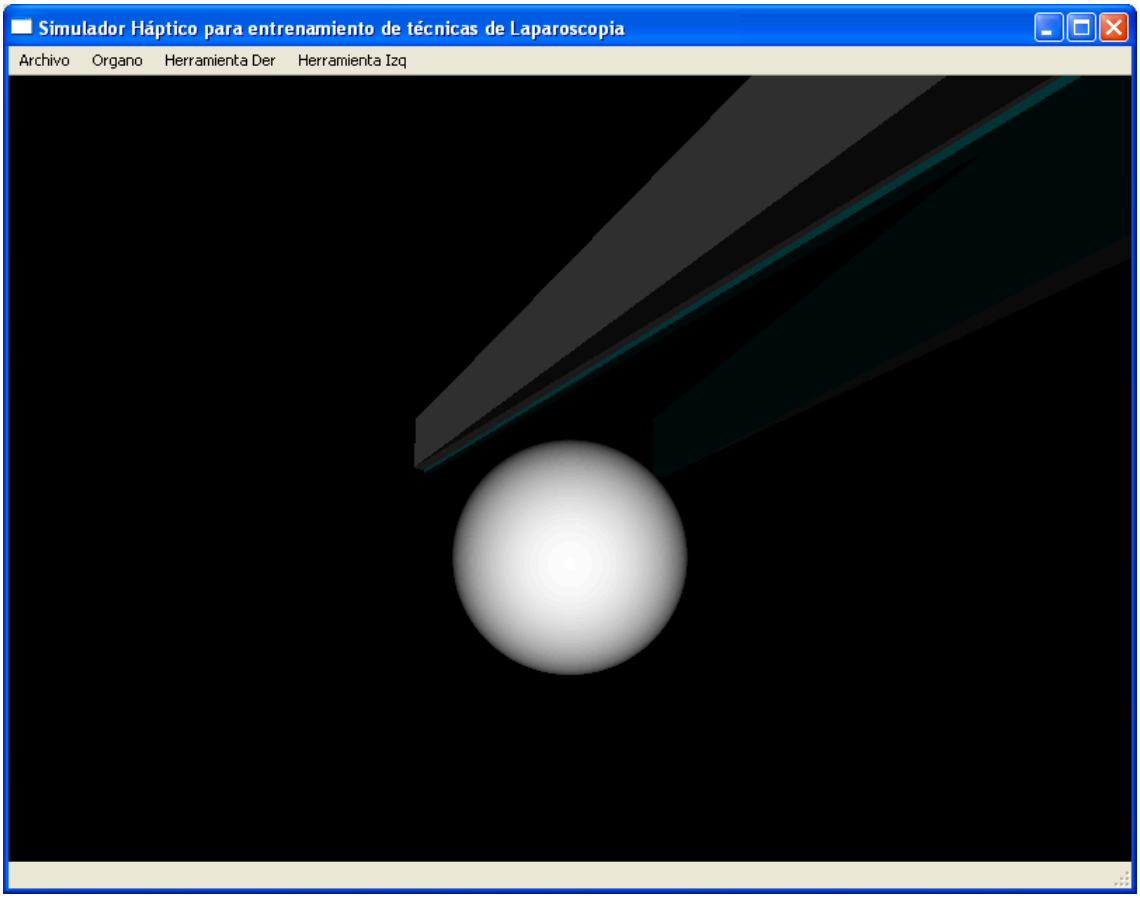
El cutter es parecido al lapiz, aunque para pintar no sólo tiene que colisionar sino que además el usuario tiene que hacer clic en el botón izquierdo del ratón.



Las tijeras son una herramienta articulada, que permite rotar sobre su eje, abrir y cerrar las tijeras. Por tanto, cabe distinguir dos acciones: rotar y abrir/cerrar. Para cambiar de acción se tiene que pulsar ‘a’. Para realizar la acción en un sentido se ha de hacer clic en el botón izquierdo del ratón. Para realizar la acción en sentido opuesto se ha de hacer clic en el botón derecho.

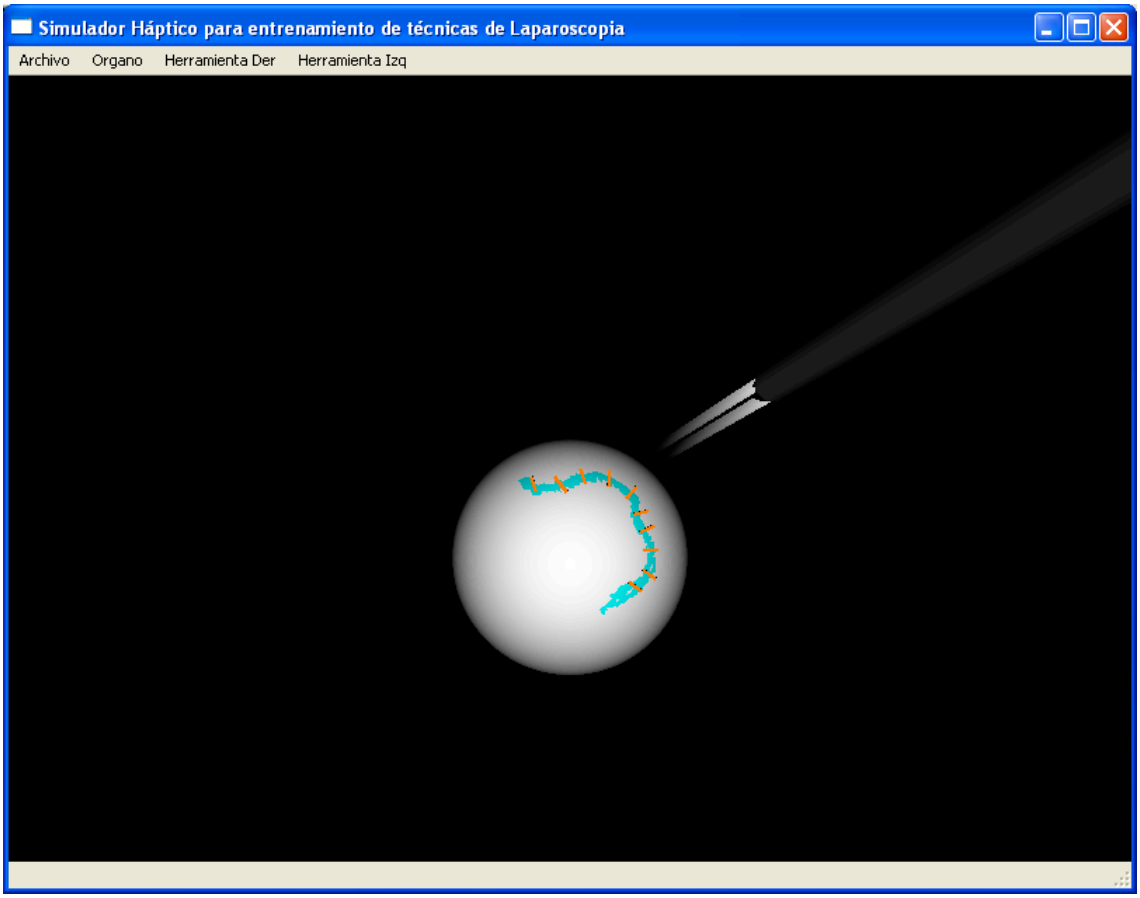


Las pinzas son una herramienta fija que permite abrir y cerrar las pinzas con los botones izquierdo y derecho del ratón respectivamente.



La grapadora es una herramienta articulada, que permite rotar sobre su eje, abrir y cerrar las pinzas y grapar. Por tanto, cabe distinguir tres acciones: rotar, abrir/cerrar y grapar. Para cambiar de acción se tiene que pulsar ‘a’. Para realizar la acción en un sentido se ha de hacer clic en el botón izquierdo del ratón. Para realizar la acción en sentido opuesto se ha de hacer clic en el botón derecho.

En la imagen que se muestra a continuación se observa la grapadora, despues de haber colocado unas cuantas grapas. La diferente orientación de las grapas se consigue gracias a la capacidad de rotación de la parte articulada de la grapadora (Ensamblado2) sobre su propio eje.



Tal y como muestra la imagen la aplicación permite la simulación de grapado a través de una trayectoria previamente definida.

Capítulo 7. Conclusiones y trabajo futuro.

Conclusiones

El principal objetivo que se pretendía lograr y que personalmente creo que se ha logrado ha sido la obtención de una aplicación de simulación en la que se modelen diferentes herramientas de laparoscopia y en la que además se implemente un sistema de detección colisiones a nivel de objeto.

A nivel personal debo decir que al tratarse de un proyecto creativo, ha resultado una experiencia gratificante y que me ha permitido conocer dos librerías muy útiles para la programación gráfica como son VTK y QT.

Como última conclusión, me gustaría acabar diciendo que considero muy acertada la elección de un proyecto fin de carrera de este tipo para la obtención del título de Ingeniero de Telecomunicación. Tanto por el tipo de proyectos que oferta el departamento de robótica y visión por computador de area de Ingeniería y Sistemas de la Universidad Miguel Hernández de Elche como por la amabilidad de los profesores, si alguien me pidiera su opinión sobre si es o no aconsable realizar el proyecto con este departamento, sin duda la respuesta es sí..

Como punto final de este apartado se presentan las características actuales del simulador:

- permite utilizar más de un organo en el escenario
- permite utilizar dos herramientas, una situada en la derecha y otra situada en la izquierda del escenario.
- Detección de colisiones entre órganos-herramientas y herramienta-herramienta
- Permite pintar sobre un órgano
- Permite la colocación de grapas orientadas según la dirección de la grapadora
- Permite movimiento en azimuth y elevación de las herramientas
- Permite articular movimientos rotacionales de las herramientas
- Permite articular movimientos de apertura/cierre de pinzas.

- Permite la simulación de grapado a través de una trayectoria previamente definida

Trabajo Futuro

Aunque este proyecto cierra una etapa importante de mi carrera, desde luego, de lo que no hay lugar a dudas, es que dentro de la rama médica en la que se ha elaborado el proyecto abre paso a una gran cantidad de proyectos:

Posibles ideas o trabajos a seguir tomando de base este proyecto podrían tratar sobre:

- Implementación de órganos deformables
- Implementación de nuevas herramientas de laparoscopia, como por ejemplo, herramientas de sutura o herramientas de cauterización en forma de garfio, etc..
- Implementación de nuevas herramientas para el aprendizaje: borrador, regla, etc..
- Ejercicios de simulación diferentes al de grapado a lo largo de una trayectoria como por ejemplo: ejercicios de penetración, ejercicios de agarre, etc..
- Ejercicios evaluativos/estadísticos de la precisión y del nivel de objetivos alcanzados
- Integración de las herramientas de laparoscopia con dispositivos hápticos
- Estudio para la optimización de colisiones
- Simulación de la gravedad

Por otro lado, para otros proyectos que necesiten la creación de nuevos objetos en VTK, su manipulación, o que necesiten integrar un sistema de detección de colisiones, tambien les puede ser de gran utilidad consultar la documentación y/o el código fuente de este proyecto así como los anexos.

ANEXOS

Anexo A: Configuración del entorno para depurar

A continuación se muestra una miniguía de gran utilidad para instalar un entorno de trabajo con QT, VTK y Visual Studio 2005 en un equipo con sistema operativo Windows XP. El último paso explica brevemente cómo instalar las librerías V-Collide.

La información mostrada a continuación surge de mi propia experiencia (muchas horas de intentar que el conjunto de aplicaciones funcionara…) y de las siguientes URLs:
http://qtnode.net/wiki/Qt4_with_Visual_Studio
http://code.hkpeterpeter.googlepages.com/vtk_build_vc.pdf

Las URLs anteriores incluyen enlaces para la descarga directa de las librerías y del parche.

La configuración del entorno requiere las siguientes operaciones (el orden importa):

1) Paso 1: Instalar y parchear QT
2) Paso 2: Descargar CMake
3) Paso 3: Instalación de VTK
4) Paso 4: Instalación de librerías V-Collide y RAPID (para detección de colisiones)

Paso 1: Instalar y parchear QT

Para poder trabajar con QT y proyectos de Visual Studio hace falta utilizar una versión comercial. Sin embargo, en Internet existe un parche para poder utilizar QT Open Source desde Visual Studio 2005.

Estos son los pasos que yo he seguido:

descargar y descomprimir qt-win-opensource-src-4.2.3.zip de trolltech.com en:
C:\Qt\4.2.3

descargar acs-4.2.3-patch1.zip de sourceforge, es el parche para que la version open source de Qt funcione para Visual Studio 2005. Descomprimirlo en C:\Qt\4.2.3

`Inicio->Todos los programas->MS V. Studio 2005->Visual Studio Tools->Command Prompt`

`cd \Qt\4.2.3` (escribir sólo la inicial de cada carpeta y completar con tabulador como si estuvieramos en linux….)
`installpatch42.bat`

A continuación si queremos una configuración en modo release, modo debug, o ambos. Aconsejo utilizar el modo debug o modo ambos, ya que depurar puede resultar de gran ayuda para encontrar errores de código.

A continuación debemos elegir una de estas tres configuraciones:
- `qconfigure msvc2005 –release`
- `qconfigure msvc2005 –debug –no-stl`
- `qconfigure msvc2005 –debug-and-release –no-stl`

Mi consejo es elegir la última, aunque debo advertir que tarda bastante tiempo.
(contestar a todo ‘y’)
`nmake` (puedes ir a hacerte un café…en mi portátil Centrino 1GB 1,6 Ghz tarda media hora para el modo release)

Desde Panel de Control, Sistema, Opciones avanzadas, Variables de entorno:
Añadir al Path: C:\Qt\4.2.3\bin
Crear una nueva variable de entorno: QMAKESPEC=win32-msvc2005

Reiniciar el equipo.

Para poder crear proyectos que utilicen QT para Visual Studio:
qmake -project -t vcapp -o projectname.pro

Desde Visual Studio 2005:

Tools->Options->Projects and Solutions->VC++ directories
C:\Qt\4.2.3\include (en include files)

Paso 2: Descargar Cmake

En mi caso, he descargardo cmake-2.4.6-win32-x86.zip y lo he descomprimido dejando un acceso directo a CmakeSetup en el escritorio.

Paso 3: Instalación de VTK

Descargar VTK, en mi caso vtk-5.0.3.rar y descomprimir en c:
Ejecutar cmakesetup y marcar show advanced values.
Source: C:\vtk-5.0.3\VTK
Binaries: C:\vtk-5.0.3\VTK_cmake (esta carpeta la creará cmake)
Configure
Nos preguntara IDE: Microsoft Visual Studio 2005

Mis opciones:
BUILD_EXAMPLES: ON
VTK_USE_GUISUPPORT: ON
El resto por defecto y Configure
USE_QVTK: ON
El resto por defecto y Configure
Así hasta que aparezca todo en gris, entonces OK

Abrir C:\vtk-5.0.3\VTK_cmake\VTK.sln
Seleccionar Release en la lista desplegable de Solution Configurations
Seleccionar ALL_BUILD project y Build
Seleccionar INSTALL project y Build
Por defecto se instala en c:\Archivos de Programa\VTK

Tools->Options->Projects and Solutions->VC++ directories (desde Visual Studio 2005):
C:\Archivos de programa\VTK\include\vtk-5.0 (en include files)
C:\Archivos de programa\VTK\lib (en library files)

Paso 4: Instalación de V-Collide

A continuación se explica brevemente los pasos para instalar V-Collide:

Descargar y descomprimir V-Collide. Puede descargarse gratuitamente de http://www.cs.unc.edu/~geom/V_COLLIDE/request.html

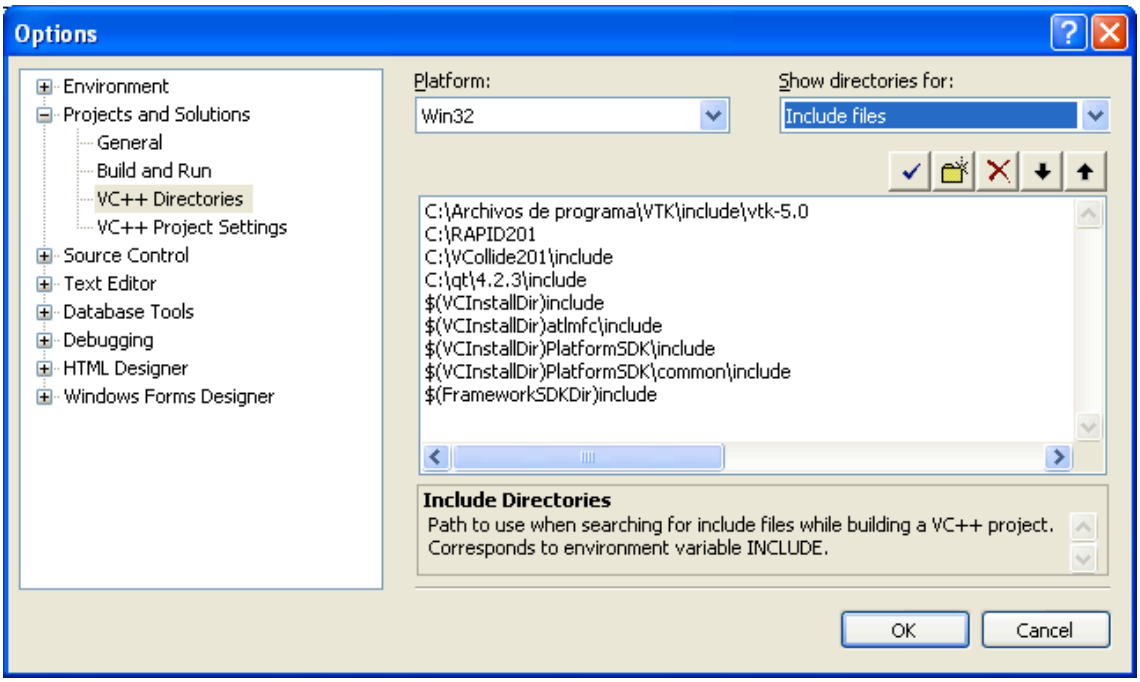
Abrir Vcollide.dsw con Visual Studio 2005 y seguir los pasos para convertir a .sln

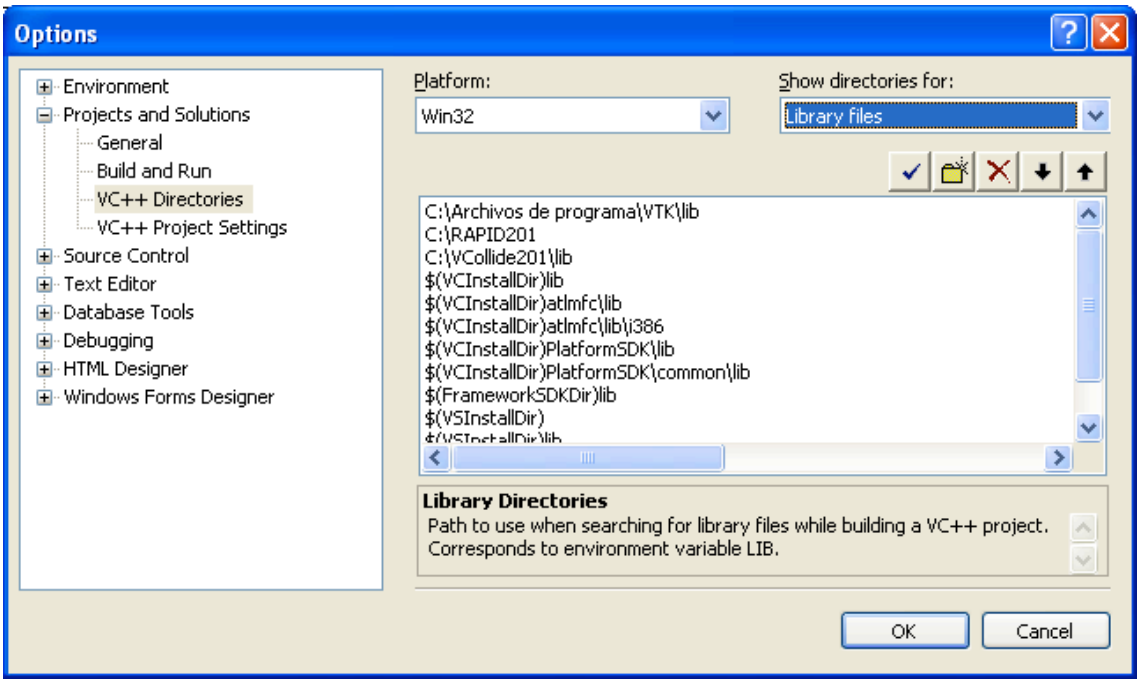
En Vinternal.c cambiar iostream.h por iostream. Hacer lo mismo con PairData.c y Nbody.c.

En Vcollide/Rapid/Rapid_private.h existe una declaración "friend RAPID_Collide(….". Añadir la palabra int entre friend y RAPID_Collide.

Por último, hay que añadir los directorios include y lib de Vcollide al IDE Visual Studio.

Para finalizar, muestro unas capturas de la configuración de Visual Studio.





Por último, para crear un proyecto de este tipo hay que tener en cuenta los siguientes pasos:

Creación de una carpeta con los fuentes y con un archivo CmakeLists.txt.

A modo de ejemplo, se muestra el contenido del archivo CmakeLists.txt utilizado en este proyecto:

```
#Indica cual debe ser el nombre del workspace resultante
PROJECT(Proyecto23mayo)


#Al incluir el siguiente código se utiliza una funcionalidad de Cmake
#para encontrar la librería VTK y en el caso de no encontrarla emite un mensaje de error
y
#se solicita que se indique el directorio donde se encuentra.
FIND_PACKAGE(VTK)
IF(NOT VTK_DIR)
  MESSAGE(FATAL_ERROR "Indica donde se encuentra el directorio de VTK
VTK_DIR")
ENDIF(NOT VTK_DIR)
INCLUDE(${VTK_USE_FILE})

# Se indica donde se encuentra los ejecutables propios de QT
# ----------MOC----------
```

```
SET(QT_MOC_EXECUTABLE ${VTK_QT_MOC_EXECUTABLE} CACHE
FILEPATH "")
# -----------UIC----------
SET(QT_UIC_EXECUTABLE ${VTK_QT_UIC_EXECUTABLE} CACHE
FILEPATH "")
# -----------QMAKE----------
SET(QT_QMAKE_EXECUTABLE ${VTK_QT_QMAKE_EXECUTABLE} CACHE
FILEPATH "")
# -----------????----------
SET(DESIRED_QT_VERSION ${VTK_DESIRED_QT_VERSION} CACHE
FILEPATH "")
#SE DETECTA LA VESION QUE ESTA INSTALADA
FIND_PACKAGE(Qt)
IF(QT_USE_FILE)
  INCLUDE(${QT_USE_FILE})
ELSE(QT_USE_FILE)
  SET(QT_LIBRARIES   ${QT_QT_LIBRARY})
ENDIF(QT_USE_FILE)

#Se indica donde esta cada una de las fuentesy de que tipo se trata
SET (PruebasSrcs  main.cxx)

#Se incluye directorios para compilar el código fuente. Este
#comando es heredado por cualquier subdirectorio. En el caso de que
#Cmake descienda a través de una estructura de directorios
#siguen estando habilitados

#Añadimos los directorios de QT e incluimos dos directorios de CMake necesarios para
el
#proceso.
INCLUDE_DIRECTORIES(
  ${QT_INCLUDE_DIR}
  ${CMAKE_CURRENT_BINARY_DIR}
  ${CMAKE_CURRENT_SOURCE_DIR}
)

IF(DESIRED_QT_VERSION MATCHES 4)
#Todas las fuentes que se quieran utilizar deben de ser incluidas
#en la lista apropiada.

#Ficheros fuente
  SET(PruebasSrcs ${PruebasSrcs}
        Principal/Principal.cpp
        Principal/Organo.cpp
        Principal/Lapiz.cpp
        Principal/LinePlotter.cpp
        Principal/MyVTKLinePlotter.cpp
        Principal/MyVTKPointPlotter.cpp
        Principal/Cutter.cpp
        Principal/Pinza.cpp
```

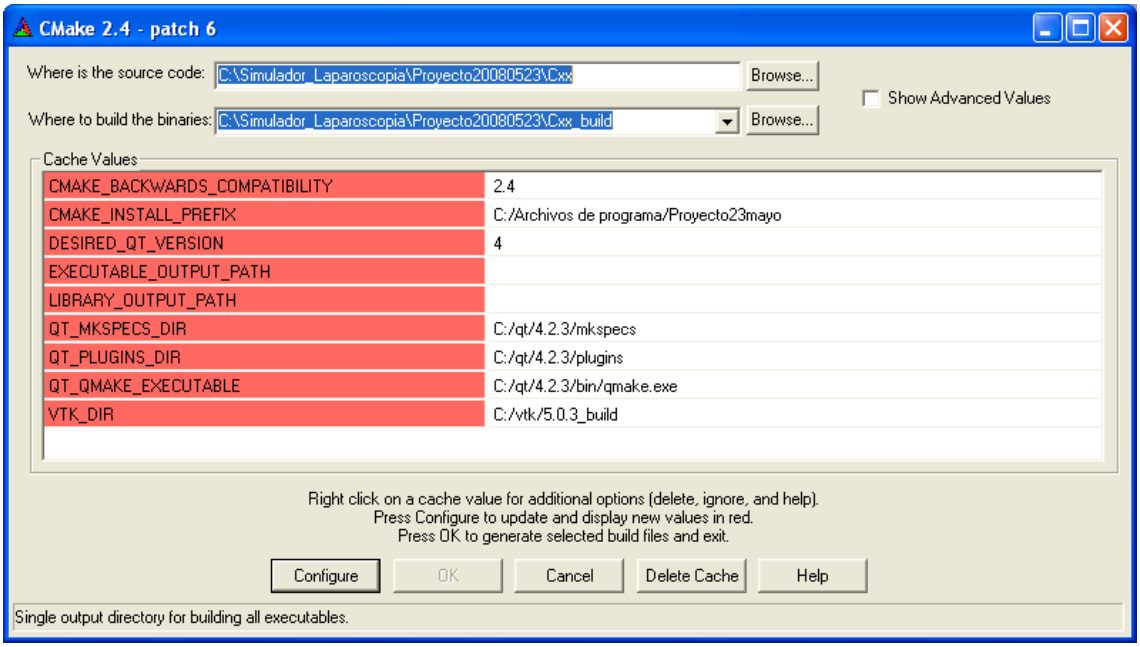
```
        Principal/Grapadora.cpp
        Principal/Tijeras.cpp
        Principal/Herramienta3D.cpp
        Principal/MyInteractorStyle.cpp
        Principal/CollisionObjectFactory.cpp
        )

#Ficheros de interfaz
SET(UIS
        Principal/Ventana_principal.ui
        )
#Cabeceras
  SET (PruebasHeaders
        Principal/Principal.h
        Principal/Organo.h
        Principal/Lapiz.h
        Principal/LinePlotter.h
        Principal/MyVTKLinePlotter.h
        Principal/MyVTKPointPlotter.h
        Principal/Cutter.h
        Principal/Pinza.h
        Principal/Grapadora.h
        Principal/Tijeras.h
        Principal/Herramienta3D.h
        Principal/MyInteractorStyle.h
        Principal/CollisionObjectFactory.h
        )


#definiciones finales del funcionamiento (donde van
#los ficheros obtenidos de la operacion UIC y MOC)
#Se indican los ficheros que se deben procesar por el motor de Qt. En la primera línea
se
#indican los ficheros UIs que serán procesados por el procesador UIC (User Interfaz
#Compiler) mientras que en la segunda lína se indican los ficheros Header a procesaro
por
#el sistema MOC (Meta Object Compiler).
  QT4_WRAP_UI(UIHeaders ${UIS})
  QT4_WRAP_CPP(MOCSrcs ${PruebasHeaders} )

#Es necesario incluir la definición de las librerías Qt e indicar las dependencias entre las
#cabeceras (UIHeaders) y los ficheros fuente (VT_proyectoSrcs) para que los
compiladores
#mencionados anteriormente (MOC y UIC) puedan trabajar correctamente.
  ADD_DEFINITIONS(-DQT_GUI_LIBS -DQT_CORE_LIB -DQT3_SUPPORT)
#Se indica la dependencia entre las cabeceras UIHeaders y los Fuentes de Qt
  SET_SOURCE_FILES_PROPERTIES(${PruebasSrcs} PROPERTIES
                        OBJECT_DEPENDS "${UIHeaders}")

ELSE(DESIRED_QT_VERSION MATCHES 4)
```

```
ENDIF(DESIRED_QT_VERSION MATCHES 4)

#Es muy cómodo que aparezca un listado de los ficheros .ui en las diferentes listas de
#ficheros de los compiladores ya que permitirá con un doble click abrir el QtDesigner
para
#su modificación. Para ello es necesario incluir el siguiente código en el proyecto.

IF (${CMAKE_BUILD_TOOL} MATCHES "msdev")
  SET (PruebasSrcs ${PruebasSrcs} ${UIS})
ENDIF (${CMAKE_BUILD_TOOL} MATCHES "msdev")
IF (${CMAKE_BUILD_TOOL} MATCHES "devenv")
  SET (PruebasSrcs ${PruebasSrcs} ${UIS})
ENDIF (${CMAKE_BUILD_TOOL} MATCHES "devenv")

#Parte propia de cualquier proyecto VTk donde además de indicar el nombre
# y tipo de la salida se incluyen
#los tres paths de ficheros fuente
ADD_EXECUTABLE( Proyecto23mayo MACOSX_BUNDLE ${PruebasSrcs}
${UISrcs} ${MOCSrcs}       ${PruebasHeaders} )

#Dentro de las librerias objetivo(target) se incluye tanto la del proyecto Pruebas
#como la reseña de la libreria QVTK
TARGET_LINK_LIBRARIES( Proyecto23mayo
  QVTK
  ${QT_LIBRARIES}
  vtkRendering
  vtkGraphics
  vtkIO
  vtkCommon
  vtkHybrid
  vtkImaging
  vtksys
  vtkFiltering
  vtkVolumeRendering
)
```

A continuación, y suponiendo por ejemplo, que la ruta en la que se encuentran el archivo CmakeLists.txt y el código fuente es la siguiente: C:\Simulador_Laparoscopia\Proyecto20080523\Cxx

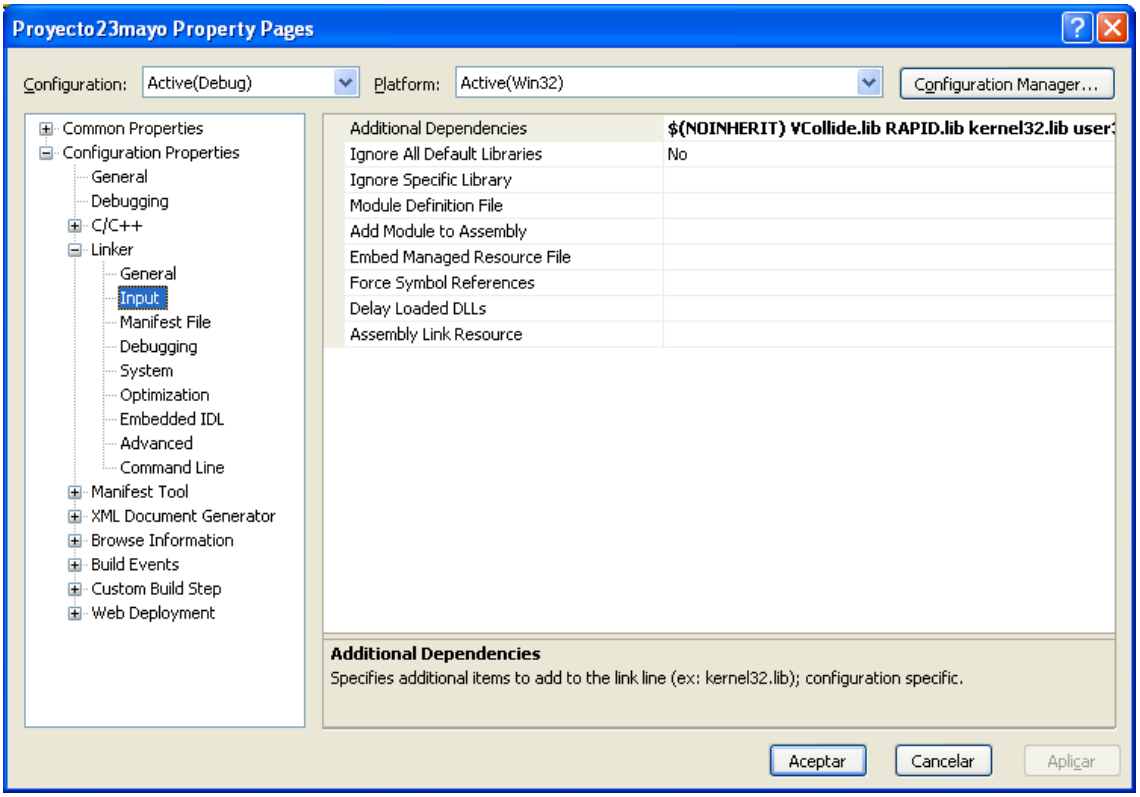
Crear una carpeta C:\Simulador_Laparoscopia\Proyecto20080523\Cxx_build

Ejecutar Cmake:



Una vez haya acabado su tarea Cmake, abrir el archivo .sln generado por Cmake con Visual Studio 2005.

Seleccionar Menu Projects, Properties (o Alt+F7) y añadir Vcollide.lib y RAPID.lib, tal y como se muestra en la siguiente imagen.



A partir de aquí ya se puede compilar.

Anexo B: Integración de VTK y VCollide

Para integrar VTK y VCollide en el proyecto de Laparoscopia se ha desarrollado una clase intermediaria entre VTK y VCollide: CollisionObjectFactory.

En la clase CollisionObjectFactory, de ahora en adelante COF, se ha desarrollado un método fundamental para añadir un objeto VTK al sistema VCollide, el método AddObject.

Este método se encarga de hacer el siguiente procesado con el objeto que se le pasa como parámetro:

- triangularización de los polígonos
- escalado en los ejes X,Y y Z
- obtención de información de los triángulos (coordenadas de los vértices y asociaciones de vértices) y almacenamiento en matrices dinámicas.
- creación de un objeto Vcollide utilizando para ello la información de los triangulos almacenada en las matrices dinámicas.

La definición del método AddObject:

```
// Añade objeto al sistema VCollide y almacena el indice del array
PiezasArray
      void AddObject(int* id,vtkAlgorithm *algorithm, double
scaleX,double scaleY,double scaleZ,char* nombre, vtkActor* actor, int*
PA_indice);
```

Veamos a continuación qué son y para que sirven cada uno de los parámetros de este método AddObject.

El parámetro id. Identificador de Vcollide. Vcollide utiliza un identificador para cada objeto añadido al sistema de colisiones. Necesita que se le pase un puntero a entero donde almacenar el identificador que le corresponda. Se trata de una asignación ascendente, empieza en cero y no se resetea nunca.

El parámetro algorithm. Se trata del objeto que queremos añadir al sistema de colisiones y puede ser cualquier tipo de objeto fuente disponible en vtk como por ejemplo un vtkSphereSource, vtkConeSource, etc.. así como un vtkOBJReader, si hemos obtenido

el objeto a partir de un archivo .obj (utilizado por ejemplo para obtener las piezas que componen las herramientas de laparoscopia).

Los parámetros scaleX, scaleY y scaleZ.

Para el que ya conozca VCollide, sabrá que tiene ciertas limitaciones o requisitos como por ejemplo que no soporta escalado de objetos y que requiere que los polígonos de los objetos sean triángulos. Estas limitaciones no hay que tenerlas en cuenta ya que AddObject, previamente a su interacción con VCollide, incluye un filtro de triangularización y un filtro de escalado (parámetros scaleX,scaleY,scaleZ).

Los parámetros scaleX, scaleY y scaleZ sirven para escalar el objeto en los ejes X,Y y Z respectivamente.

Los posibles valores que pueden tomar estos parámetros de escalado son:

- 1, indica que no hay escalado
- 0<x<1, indica una reducción del tamaño
- >1, indica que un aumento del tamaño

El parámetro nombre. Sirve para asignar un nombre a la pieza colisionable.

El parámetro actor. El actor que representa a la pieza colisionable.

El parámetro PA_indice. Índice del array PiezaArray. Para llevar el control de los objetos colisionables que se están utilizando en la aplicación se ha creado un Array dinámico de tamaño TAM_MAX_PIEZAS_ARRAY: PiezasArray, de ahí el primer término de este parámetro: PA. De igual forma que el parámetro id, el valor del índice se asignará desde el propio método AddObject.

```
	// Array con información sobre las piezas añadidas al sistema de colisiones
	ObjectInfo* PiezasArray;

// Structura de datos sobre cada objeto/pieza del sistema de colisiones
	struct ObjectInfo {
		int VC_Id; // ID en VCollide
		char nombre[50]; // Nombre de la pieza
		vtkActor* actor; //actor correspondiente a la pieza
```

```
            vtkTransformPolyDataFilter* filtro; //filtro
(triangularizado y escalado)
            bool isOrgano; //diferencia organos y herramientas
      };
```

Por último, se procede como es habitual en VTK, es decir, creación de un mapper, asociación del mapper al actor y por último se añade el actor al Render.

Es muy importante tener en cuenta que el mapper necesita el filtro del objeto triangularizado y escalado. Para obtenerlo basta utilizar el método GetFilterOfPiezasArray de la clase CollisionObjectFactory pasándole como parámetro el indice del array PiezasArray.

Ejemplo de creación de un objeto colisionable.

Veamos un ejemplo de cómo crear un objeto colisionable, en este caso un cono:

```
// Creacion un objeto CollisionObjectFactory
CollisionObjectFactory COF = new CollisionObjectFactory();

//creacion de un cono en vtk
vtkSphereSource *cone = vtkSphereSource::New();
cone->SetRadius(50.0);
cone->SetPhiResolution(100);
cone->SetThetaResolution(100);

vtkActor *coneActor;
int PA_id;
int id;

// añadir objeto al sistema de colisiones
COF->AddObject(&id, cone,1,1,1,"Organo",coneActor,&PA_id);

vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();

//Si el objeto no fuera colisionable: cone->GetOutputPort()
//coneMapper->SetInputConnection( cone->GetOutputPort() );
//

// Para objeto colisionable obtenemos el filtro de COF
coneMapper->SetInputConnection(COF->GetFilterOfPiezasArray(PA_id)-
>GetOutputPort());

coneActor = vtkActor::New();
coneActor->SetMapper( coneMapper );
coneActor->GetProperty()->SetDiffuseColor(0.6,0.6,0.1);
ren->AddActor( coneActor );
```

Las diferencias respecto a crear un objeto “no colisionable” son las siguientes líneas:

```
COF->AddObject(&this->id, cone,1,1,1,"Organo",coneActor,&PA_id);
```

```
coneMapper->SetInputConnection(COF->GetFilterOfPiezasArray(PA_id)-
>GetOutputPort());
```

donde id y PA_id son variables tipo Int que sirven para identificar al objeto en VCollide y en el sistema de colisiones (CollisionObjectFactory) respectivamente. Estas variables no hay que inicializarlas ya que es dentro del propio método AddObject desde donde se les asignará valor.

El método AddObject se encarga de procesar el objeto (triangularizar y escalar antes de añadir al sistema de colisiones) y de guardar el filtro con el procesamiento realizado.

```
void AddObject(int* id,vtkAlgorithm *algorithm, double scaleX,double
scaleY,double scaleZ,char* nombre, vtkActor* actor, int* PA_indice);
```

Para recuperar el filtro utilizaremos el método GetFilterOfPiezasArray:

```
vtkTransformPolyDataFilter* GetFilterOfPiezasArray(int PA_index) {
return this->PiezasArray[PA_index].filtro; }
```

El método AddObject:

```
void CollisionObjectFactory::AddObject(int *id,vtkAlgorithm
*algorithm, double scaleX, double scaleY, double scaleZ, char* nombre,
vtkActor* actor, int* PA_indice)
{
	// Para añadir un objeto al sistema de colisiones VCollide hay
	// que tener en cuenta dos consideraciones:
	// 1) Triangularizar
	// 2) No soporta escalado, por lo que el escalado debe hacerse
	//    a priori utilizando vtkTransformPolyDataFilter

	//std::cout << "Añadiendo objeto al sistema de colisiones\n";
	// Triangularizamos
	vtkTriangleFilter* triObject = vtkTriangleFilter::New();
	triObject->SetInputConnection(algorithm->GetOutputPort());
	triObject->SetPassLines(5);

	// Escalamos
	vtkTransform *tr = vtkTransform::New();
	tr->Scale(scaleX,scaleY,scaleZ);

	vtkTransformPolyDataFilter *algorithmFiltered =
vtkTransformPolyDataFilter::New();
	algorithmFiltered->SetInputConnection(triObject-
>GetOutputPort());
	algorithmFiltered->SetTransform(tr);

	// para que getpoints no devuelva cero
	triObject->Update();
	algorithmFiltered->Update();

	vtkPoints *puntos = vtkPoints::New();
	puntos=algorithmFiltered->GetOutput()->GetPoints();
```

```
      //std::cout << "matriz dinámica de " << puntos-
>GetNumberOfPoints() << " puntos" << std::endl;

      // creamos matriz dinamica para almacenar coordenadas de los
puntos
      double **matriz;
      matriz = new double* [puntos->GetNumberOfPoints()];

      for (int i=0;i<puntos->GetNumberOfPoints();i++)
        {
        matriz[i]=new double[3];
        }

      // almacena las coordenadas de los puntos en la matriz dinámica
      for (int i=0;i<puntos->GetNumberOfPoints();i++)
        {
              puntos->GetPoint(i,matriz[i]);
        }

      // crea un nuevo objeto VCollide y lo prepara para añadirle
triangulos
      vc.NewObject(id);

      //Despues de llamar a NewObject VCollide ya ha asignado un id
      int numObj=*id;

      cout << "COF ha asignado para un nuevo objeto el id=" <<
numObj<<"\n";

      // Almacena información sobre el objeto en PiezasArray
      *PA_indice = this->SetDataOfObjectInPiezasArray(numObj, nombre,
actor, algorithmFiltered);

      cout << "COF ha asignado para un nuevo objeto el id interno=" <<
*PA_indice<<"\n";

      //añadimos al vector de objetos un puntero al objeto ya filtrado
(escalado y triangularizado)
      //this->objetos[numObj]=algorithmFiltered;

      vtkCellArray *celdas = vtkCellArray::New();
      celdas=algorithmFiltered->GetOutput()->GetPolys();

      //identificador de triangulo
      int idTri=0;

      vtkIdType npts;
      vtkIdType *pts;

      //añade los triangulos en base a la informacion de las celdas
      while (celdas->GetNextCell(npts,pts)!=0)
        {
        vc.AddTri(matriz[pts[0]],matriz[pts[1]],matriz[pts[2]],idTri);
        idTri++;
        }

      vc.EndObject();

      //std::cout << "Objeto añadido OK al sistema de colisiones\n";
```

```
	// se destruye la matriz dinamica
	for (int i=0;i<puntos->GetNumberOfPoints();i++)
		{
		delete [] matriz[i];
		}
	delete [] matriz;

	//std::cout << "Matriz dinamica destruida\n";


	tr->Delete();
	//algorithmFiltered->Delete();	//celdas y puntos no hace
falta
	triObject->Delete();
	cout << "Fin de AddObject\n";
}
```

El método `SetDataOfObjectInPiezasArray`

```
// rellena un elemento del array PiezasArray y devuelve el indice en
el array o -1 (error)
	int SetDataOfObjectInPiezasArray(int VC_Id, char* nombre,
vtkActor* actor, vtkTransformPolyDataFilter* filtro);
```

Por último, destacar que V-Collide no entiende de distancias, simplemente detecta si hay o no colisión.

En el caso de colision, para saber qué triángulos han colisionado es muy importante el buen conocimiento de la clase VCReport de VCollide.

La clase VCReport consta de los siguientes métodos:

`int numObjPairs();`	Número de pares de objetos en contacto
`int obj1ID( int obj_pair_num );`	ID del primer objeto especificado en el par de objetos
`int obj2ID( int obj_pair_num );`	ID del segundo objeto especificado en el par de objetos
`int numTriPairs( int obj_pair_num );`	Devuelve el número de pares de triángulos que están en contacto entre los pares de objetos especificados. (útil sólo cuando se llama a Vcollide::Collide() con VC_ALL_CONTACTS.)
`int tri1ID( int obj_pair_num, int tri_pair_num );`	ID del primer triángulo en el par de triángulos específicado. Tri_pair_num va desde 0 hasta el número de pares de triángulos - 1
`int tri2ID( int obj_pair_num, int tri_pair_num );`	ID del segundo triángulo en el par de triángulos específicado. Tri_pair_num va desde 0 hasta el número de pares de triángulos - 1

A modo de ejemplo, el siguiente código se utiliza por la herramienta Cutter para obtener qué triángulos tiene que colorear:

```
void Cutter::clicIzq()
{
	this->OCF->InformAboutCollisions();
	// obtiene el id del organo colisionado
	int organo_PA_Id=this->OCF-
>GetOrganoCollisionedWithHerramienta(this->ActorsVCIdArray[1]);
	// comprueba si ha colisionado el objeto invisible y si ademas
ha colisionado con un organo
	if (this->OCF->HasCollisioned(this-
>ActorsVCIdArray[1])&&organo_PA_Id!=-1)
	 {
		//tiene que pintar
		int numCeldas;
		scalars=vtkUnsignedCharArray::SafeDownCast(this->OCF-
>GetFilterOfPiezasArray(organo_PA_Id)->GetOutput()->GetCellData()-
>GetScalars());
		if (scalars==NULL)
			{
			numCeldas=this->OCF-
>GetFilterOfPiezasArray(organo_PA_Id)->GetOutput()-
>GetNumberOfCells();
			std::cout << "generando scalars..para numceldas=" <<
numCeldas << std::endl;
			scalars=vtkUnsignedCharArray::New();
			scalars->SetNumberOfComponents(4);
			scalars->SetNumberOfTuples(numCeldas);
			float r=0;float g=0;float b=0;
			for (int i=0;i<numCeldas;i++)
				{
				scalars->SetTuple4(i,255,255,255,255);
				}
			this->OCF->GetFilterOfPiezasArray(organo_PA_Id)-
>GetOutput()->GetCellData()->SetScalars(scalars);

			}
		this->OCF->GetFilterOfPiezasArray(organo_PA_Id)-
>GetOutput()->GetCellData()->Modified();
		//PINTA
		int obj_pair_num=this->OCF-
>GetCollisionedPairOfObjectsNum(this->ActorsVCIdArray[1]);
		int num_tri;
		// para saber si se ha de colorear tri1ID o tri2ID
		int obj_id=this->OCF->report.obj1ID(obj_pair_num);
		bool tri1=false;
		if (obj_id==organo_PA_Id) tri1=true;
		else tri1=false;

		for (int k=0;k<this->OCF-
>report.numTriPairs(obj_pair_num);k++)
			{
				if (tri1==true) {
					num_tri=this->OCF-
>report.tri1ID(obj_pair_num,k);
				}
				else {
					num_tri=this->OCF-
>report.tri2ID(obj_pair_num,k);
				}

				}// pinta de color rojo
```

```
                        scalars->SetTuple4(num_tri,220,17,17,255);
                        }
            this->OCF->GetFilterOfPiezasArray(organo_PA_Id)-
>GetOutput()->GetCellData()->Modified();
            }
      else std::cout << "No puede pintar.\n";
```

Anexo C: Manual de usuario de V-Collide

```
V-COLLIDE USER'S MANUAL - Release 2.01
       ---------------------------------------


INTRODUCTION

    The V-Collide collision detection library performs efficient and
    exact collision detection between triangulated polygonal models.
It
    uses a 2-level hierarchical approach: the top level eliminates
from
    consideration pairs of objects that are not close to each other,
    while the bottom level performs exact collision detection down to
    the level of the triangles themselves.

    The basic steps involved in using this library are creating
objects,
    adding sets of triangles to these objects, choosing which pairs of
    objects should be tested for collisions, setting the positions of
    the objects, performing the collision test, and getting back
reports
    of the test results.  Based on these results and any other
    parameters of the simulation/interaction, the objects may be moved
    and the collisions tested again, etc.

    V-Collide is written in C++, and, unlike previous versions,
    Release 2.01 no longer provides C interface.

C++ COMMAND REFERENCE

    VCollide Class
    --------------
    #include <VCollide.H>

    int VCollide::NewObject (int *id);
      Create a new object and prepare it for adding triangles.

    int VCollide::AddTri (double v1[3], double v2[3], double v3[3],
                          int tri_id = 0);
      Add triangles to the current object (only valid between
      NewObject() and EndObject()), and give an id to the triangle.
        If no triangle id is given, the default is 0.

    int VCollide::EndObject (void)
      Finish adding triangles to the current object and build the
      hierarchical collision detection structures for the object.

    int VCollide::DeleteObject (int id);
      Delete an object.

    int VCollide::ActivateObject (int id);
      Turn on collision detection for an object.

    int VCollide::DeactivateObject (int id);
      Turn off collision detection for an object.
```

```
    int VCollide::ActivatePair (int id1, int id2);
      Turn on collision detection between a specific pair of objects.

    int VCollide::DeactivatePair (int id1, int id2);
      Turn off collision detection between a specific pair of objects.

    int VCollide::UpdateTrans (int id, double trans[4][4]);
      Update the transformation applied to an object.
      Note that we consider only the change in position of the
      object. Scaling is not supported.

    int VCollide::Collide (int flag = VC_FIRST_CONTACT);
      Compute collisions. If flag is VC_FIRST_CONTACT, V-Collide
detects
        only the first intersection between each pair of objects. If
flag
        is VC_ALL_CONTACTS, V-Collide reports the ids of all the
        intersecting triangles in each pair of intersecting objects.
        Default is VC_FIST_CONTACT.

    int VCollide::Collide(VCReport *report, int flag =
VC_FIRST_CONTACT);
      Compute collisions and return a collision report. If flag is
        VC_FIRST_CONTACT, V-Collide detects only the first
intersection
        between each pair of objects. If flag is VC_ALL_CONTACTS,
        V-Collide reports the ids of all the intersecting triangles in
        each pair of intersecting objects. Default is VC_FIST_CONTACT.

    VCReport Class
    --------------
    #include <VCollide.H>

    int numObjPairs();
      Returns the number of pairs of objects that are in contact.

    int obj1ID( int obj_pair_num );
      Returns the ID of the first object in the specified object-pair.
      obj_pair_num ranges from 0 to numObjPairs() - 1.
        Error may occur if obj_pair_num is out of range.

    int obj2ID( int obj_pair_num );
      Returns the ID of the second object in the specified object-
pair.
        Error may occur if obj_pair_num is out of range.

    int numTriPairs( int obj_pair_num );
      Returns the number of pairs of triangles that are in contact
        between the specified pair of objects. This is useful only if
      VCollide::Collide() was called with VC_ALL_CONTACTS.
      If VCollide::Collide() was called with VC_FIRST_CONTACT,
        then numTriPairs() always returns 0. Error may occur if
        obj_pair_num is out of range.

    int tri1ID( int obj_pair_num, int tri_pair_num );
      Returns the ID of the first triangle in the specified
        triangle-pair. This triangle belongs to the first object in
the
        specified object-pair. Used only if VCollide::Collide() was
        called with VC_ALL_CONTACTS. Error may occur if obj_pair_num
        or tri_pair_num is out of range. tri_pair_num ranges from 0
```

to 1 less than the number of triangle-pairs.

int tri2ID(int obj_pair_num, int tri_pair_num);
Returns the ID of the second triangle in the specified triangle-pair. This triangle belongs to the second object in the specified object-pair. Used only if VCollide::Collide() was called with VC_ALL_CONTACTS. Error may occur if obj_pair_num or tri_pair_num is out of range.

OVERVIEW

V-Collide is a library that provides fast, exact collision detection, and may be useful for virtual reality and a variety of other simulations applications. It assumes that you have a model that may be broken down into a set of objects. These objects typically undergo some rigid-body motions, and are periodically tested for collision. For each simulation time step, the V-Collide library will report which pairs of objects are currently in contact. You are free to specify exactly which pairs of objects should be tested for collision, and which may be ignored.

With V-Collide 2.01, you can specify whether you want the library to just report which pairs of objects are currently in contact, or, in addition to that, which triangles are intersecting between each pair of intersecting objects.

INITIALIZATION

Users may initialize a collision detection engine by declaring a VCollide object. It is possible to run multiple independent collision detection engines by simply declaring multiple VCollide objects.

CREATING OBJECTS

To create an object, first call NewObject(), which will set up an empty object and provide an integer ID for that object.

For every triangle in the object, call AddTri() with the coordinates of its three vertices. If you want the library to report the intersecting triangles between objects (calling Collide() with VC_ALL_CONTACTS), you need to call AddTri() with an additional integer argument to identify each triangle in the object. If the object has faces with more than three vertices, you will first need to triangulate them.

When you are done adding triangles, call EndObject(). This tells the library to build its data structures for the object.

The only library call which may appear between NewObject() and EndObject() is AddTri().

Objects can be deleted with DeleteObject().

You can create or delete objects at any time during the simulation,

but building the data structures for these objects has some run-time
overhead.

MANAGING ACTIVATION STATE

V-Collide's activation state determines which objects will be tested
for collisions. The activation state has two components -- a per-object component and a pairwise component (for each possible pair of objects). These two components are managed independently, so for a pair of objects to be tested for collision, not only must the pair be active, but each of the two objects must be active as well.

When an object is created, it is active by default. In addition, all pairs of objects that include this new object are also active.

The pairwise component of the activation state is managed using the
ActivatePair() and DeactivatePair() calls, while the per-object component is managed using the ActivateObject() and DeactivateObject() calls. Because these components are managed independently, modifying an objects activation state does not affect
the activation state of an object's pairs.

(Note to I-Collide users: do not confuse the ActivateObject() call in
V-Collide with the activate_full() call in I-Collide. All the activation calls in I-Collide operate on the pairwise activation state. I-Collide does not have a per-object activation state.)

MOVING OBJECTS

To move an object, call UpdateTrans(). This function takes as arguments the ID of the object and a new transformation for it, expressed as a 4x4 matrix, formulated to be multiplied to the left of a column vector during the transformation. This matrix should be
a rigid-body transformation -- rotation and translation. When an object is first created, it has the identity matrix as its transformation.

PERFORMING THE COLLISION TEST

When all the objects' transformations have been modified as necessary and the activation state is properly set, call Collide() to perform the collision testing for the current time step. Collide() can be called with argument VC_FIRST_CONTACT (default) or VC_ALL_CONTACTS.

With VC_FIRST_CONTACT, Collide() will only detect the first contact
between each pair of objects and report which pairs of objects are currently in contact. With VC_ALL_CONTACTS, Collide() will detect all the contacts between each pair of objects and report, in addition to what you can get with VC_FIRST_CONTACT, which triangles are intersecting between each pair of intersecting objects.

GETTING REPORTS

After calling Collide() with the address of a VCReport object, the VCReport object will be filled with the collision results, and the results can be read using the VCReport's functions.

RETURN VALUES

Most operations called on a VCollide engine return an integer success code: VC_OK on success, or an appropriate error code on failure. These codes are defined in VCollide.H. Return values should be checked regularly to make sure that the application is performing correctly.

DATA FILES

V-Collide has no native data file format, so you are free to use to format of your choice. The example code reads from several different file types.

CONVERTING FROM I_COLLIDE

The high-level structure of V-Collide is similar to that of I-Collide, but the underlying basis is different, so shapes need to be described differently. V-Collide does not specify its own file format the way I-Collide did. Instead, load your objects from whatever storage format is most convenient for you. Then, specify to V-Collide every triangle in the model.

V-Collide also differs from I-Collide in its specification of activation state. V-Collide has two distinct layers of activation state, while I-Collide has only one. As objects are created in V-Collide, they default to being active objects, and all pairwise interactions are also active by default.

V-Collide does not currently support I-Collide's object instancing, dynamically-resized bounding boxes, or distance computations.

Anexo D: Bibliografía

- **http://www.vtk.org**
- **http://www.trolltech.com**
- **http://vtkblog.blogspot.com/**
- **The Visualization Toolkit User's Guide.** Kitware. Kitware, Inc publishers.
- **The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 4th Edition** Will Schroeder, Ken Martin, Bill Lorensen. Kitware, Inc. publishers
- **A review of mechanism used in laparoscopic surgical instruments.** Jonas J.B. Lim a, Arthur G. Erdman b. http://www.sciencedirect.com
- **PFC original de Jose Ignacio Reolid sobre reconstrucción 3D.** UMH.
- **A Survey of Surgical Simulation: Applications, Technology and Education.** Alan Liu, Frank Tendick, Kevin Cleary, Christoph Kaufmann
- **http://www.cs.unc.edu/~geom/V_COLLIDE/**
- **Curso práctico de programación en C y C++.** Jorge Badenas Carpio, José Luis Llopis Borrás, Oscar Coltell Simon. Colección Universitat Jaume I.
- **EndoWrist® Instruments & Accessories Catalog.** http://www.intuitivesurgical.com.
- **Vector Math for 3D Computer Graphics.** http://chortle.ccsu.edu/VectorLessons/vectorIndex.html
- **C++ Gui Programming with QT 4.** Jasmin Blanchette., Mark Summerfield. Prentice Hall.